Root Cause Analysis of Failures in Microservices through Causal Discovery

Azam Ikram[1] **Sarthak Chakraborty**[2] **Subrata Mitra**[2] **Shiv Kumar Saini**[2]
Saurabh Bagchi[1] **Murat Kocaoglu**[2]
[1]Purdue University, USA [2]Adobe Research, India
{mikram,sbagchi,mkocaoglu}@purdue.edu
{sarchakr,sumitra,shsaini}@adobe.com

Abstract

Most cloud applications use a large number of smaller sub-components (called microservices) that interact with each other in the form of a complex graph to provide the overall functionality to the user. While the modularity of the microservice architecture is beneficial for rapid software development, maintaining and debugging such a system quickly in cases of failure is challenging. We propose a scalable algorithm for rapidly detecting the root cause of failures in complex microservice architectures. The key ideas behind our novel hierarchical and localized learning approach are: (1) to treat the failure as an intervention on the root cause to quickly detect it, (2) only learn the portion of the causal graph related to the root cause, thus avoiding a large number of costly conditional independence tests, and (3) hierarchically explore the graph. The proposed technique is highly scalable and produces useful insights about the root cause, while the use of traditional techniques becomes infeasible due to high computation time. Our solution is application agnostic and relies only on the data collected for diagnosis. For the evaluation, we compare the proposed solution with a modified version of the PC algorithm and the state-of-the-art for root cause analysis. The results show a significant improvement in top-k recall while significantly reducing the execution time.

1 Introduction

Root Cause Analysis (RCA) is widely used to ensure the reliability of production systems in many domains such as IT operations [45], telecommunications [59], and medicine [40]. Modern production systems follow a network topology that consists of multiple components connected through complex dependencies. For instance, cloud applications typically follow microservice-based architecture [46]. Such applications have the benefit of a simple development process, uncomplicated maintenance, and flexible deployment [35]. A typical cloud-based application uses several microservices, each performing some small tasks [25]. Each microservice is further instrumented to monitor its state and availability which results in thousands of metrics to monitor [10]. The larger number of metrics makes it challenging to find the root cause of the failures in a timely fashion. It has been reported that it takes on average 3 hours to identify the root cause of a failure without using any automated tools [53]. Since enterprise cloud services providers have prices tied with availability guarantees, a delay in detecting the root cause of a fault can result in significant revenue loss.

Causal structure discovery-based Root Cause Analysis techniques have been used recently to find the root cause(s) of a fault in cloud applications [41, 53, 24, 47, 1, 14, 22]. The goal is to construct a graph with nodes as metrics and a directed edge between two nodes showing the direction and magnitude of the causal effect. A direct application of causal discovery algorithms like PC [47] is

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

infeasible for a microservice system due to a large number of metrics. Existing approaches reduce the number of nodes by using a feature selection approach or selecting a specific set of metrics[48, 27, 53]. An obvious problem with this approach is that the selected set of metrics might not include the root cause metric(s). In addition, the feature selection step might introduce latent variables, which renders the popularly used PC algorithm inapplicable. Furthermore, most of these approaches only rely on observational data and therefore do not utilize the potential invariance present in the interventional data to learn the underlying causal structure.

We make two crucial observations. First, a fault changes the generative mechanism of the failing node. Hence, a fault can be modeled as an intervention on the failing node, and the data during the fault period as an interventional dataset. We borrow ideas from recent work on learning causal graphs using an interventional distribution with unknown interventional targets [19, 33]. In summary, a binary indicator variable for normal and fault periods is introduced as an additional node. The target metrics of the newly introduced node are the root causes. Second, given the first observation, we do not need to learn the full graph for RCA. Using these two observations, we propose a novel hierarchical and localized causal discovery algorithm, which we call ***Root Cause Discovery (RCD)***, to detect the root cause of a failure. The algorithm pinpoints the root causes without learning the causal structure of the complete graph which allows the algorithm to run on a much larger set of metrics. We further optimize the algorithm by hierarchically exploring the data. We validate RCD on synthetic datasets and data from the Sock-shop application test bed. The results show a significant improvement in runtime and accuracy as compared to the baselines. In addition, we also apply the algorithm to three real-life failures of microservices from a large cloud service provider. RCD correctly identifies the root cause metrics in a fraction of the time than the baseline. Summary of our contributions are as follows:

- We consider failures as interventions on the root cause nodes. Doing so allows us to use distributional invariances not only within the observational, but also across observational and interventional data for learning.
- We propose a novel solution to identify the root causes of the failure using a localized hierarchical learning algorithm. Our algorithm is not only more computationally efficient, but it also requires much fewer anomalous samples compared to the existing approaches.
- We evaluate our system on synthetic datasets and data generated from a real-world application and also report our findings from a production-ready microservice-based application.

2 Background and Motivation

In this section, we provide the background of concepts that underlie our design.

Causality and Causal graphs. A variable (metric) is said to cause another variable if a change in the former induces a change in the distribution of the latter. A causal graph is used to encapsulate the causal relationships between variables using a directed acyclic graph (DAG). Every node of the DAG represents a variable, while a directed edge shows the causal relationship between the two variables: $X \rightarrow Y$ indicates that X causes Y.

Structural causal models (SCMs) are typically used to model causality between a set of random variables. Accordingly, each variable X is assigned a value based on a functional relation with a subset Pa_X of observed variables and an exogenous noise E_X as $X = f_X(Pa_X, E_X)$. One can then construct the causal graph by assigning the set of observed variables Pa_X as the parents of X for all X. Causal Bayesian networks (CBNs) similarly can be used for defining a causal model without specifying the functional relations, and instead specifying observational and interventional distributions through the so-called truncated factorization formula under the causal graph. For a formal treatment of the subject, please refer to [38].

Intervention. Intervention in a variable is the process of changing the generative mechanism of that variable. The most widely used notion of interventions are in randomized controlled trials (RCTs), and A/B tests, where we would like to understand the causal effect of a certain treatment. This type of intervention is captured by the do-operator of Pearl [38] and destructs the previous generative mechanism. For example, $do(X = x)$ forces X to take the value of x, effectively removing the role of its structural equation. This is also called a *hard intervention*, and its effect in the causal graph is to sever the incoming edges to the intervened node. On the other hand, *soft interventions*

are used to model experiments that do not completely destruct the causal mechanisms but modify them. For example, a soft intervention on X can replace the structure equation $f_X(Pa_X, E_X)$, with some $f'_X(Pa_X, E_X)$, where $f' \neq f$, which then retains the original causal graph but changes the generative mechanism and the conditional distribution i.e., $p(X|Pa_x, do(X = x)) \neq p(X|Pa_X)$. This is known as the faithfulness assumption [38].

Causal Discovery. Given a set of observations of random variables from a system, the goal of causal discovery algorithms is to find the underlying causal structure. Most constrain-based causal discovery algorithms estimate the causal graph in two phases. The first phase focuses on building a skeleton using conditional independence (CI) tests, whereas the second part estimates the orientation of the edges between the nodes based on a set of static rules. For the unfamiliar reader, we give a detailed background on the causal graph and causal discovery in the appendix.

Most causal discovery algorithms estimate the causal graph from observational data [37, 38]. However, recently there has been some work done in learning the causal graph from both observational and interventional data as it is more informative than just relying on the observational data [21, 38, 56]. Furthermore, there are a set of studies that tries to find the intervention target (node where the intervention occurred) from the observational and interventional data [19, 50].

RCA and Causal Inference. The above mentioned concepts from causal inference can be mapped to the problem of finding the root cause of a failure. Consider that a failure happens at time t on service X, then we make the following simplifications;

- At time t, a soft intervention is performed on X;
- The metric data collected from all the services till time t is an observational (normal) dataset, denoted as $\mathcal{D}$;
- The metric data collected from time t and onward makes up the interventional (anomalous) dataset, denoted as $\mathcal{D}^*$;

This mapping allows us to consider a failure as an intervention on the failing node. Considering the failures as an intervention give us the ability to leverage the existing literature for finding the interventional target and build a customized solution for RCA, which we explain next.

3 Root Cause Analysis

In this section, we formalize the problem and discuss a few challenges that are still open in detecting the root cause of the failure.

Problem Formulation. In a cloud system, the problem of finding the root cause of the failure can be formalized as follows. A microservice-based cloud application consists of a set of n microservices, $\mathcal{S} = \{s_1, \ldots, s_n\}$. With a given time interval, the monitoring tool collects at least m metrics from each of the microservices, i.e., $\mathcal{M}(i, t) = \{r_{i,1,t}, \ldots, r_{i,m,t}\}$ where $m \geq 1; \forall i \in \{1, \ldots, n\}$. Here, $\mathcal{M}(i, t)$ is a set of m metrics of microservice i at time instance t. To combine this all together, we have two time series datasets defined as $\mathcal{D} = \{\mathcal{M}(1, 1), \ldots, \mathcal{M}(n, \mathcal{T} - 1)\}$ and $\mathcal{D}^* = \{\mathcal{M}(1, \mathcal{T}), \ldots, \mathcal{M}(n, t)\}$ where $\mathcal{T}$ represents the time when the failure was first registered and t is the time when the bug was fixed. We also assume that all the columns have discrete values as the continuous values can be discretized if needed.

The problem of localizing the failure is difficult because of the high noise-to-signal ratio because of failure propagation. A failed service will affect all the parent services and therefore it becomes difficult to find the culprit. There has been some recent literature that tries to narrow down the root cause using techniques from causal inference [58, 27, 14, 22, 32, 48], however, it has a few shortcomings.

Domain knowledge and parametric assumptions. For problem simplification, most existing works make parametric assumptions that might not hold in the real-world [48, 22, 58]. For instance, consider a case where the latency of a microservice grows linearly to the CPU utilization. However, after a point, the degree of parallelism exceeds the gain of running multiple requests in parallel (because of the increasing cost of context switches), after which the latency increases exponentially with respect to the number of incoming requests. This non-linearity has been pointed out myriad times in the systems literature [61, 29, 3].

Moreover, some studies use domain knowledge to construct a causal graph [14, 22] that limits their applicability and works only in strict cases. For example, one of the rules used in CIRCA [22] is that the callee's traffic load affects the caller's latency. This statement holds only when the communication between two services is a blocking call. If we have a non-blocking call where the caller does not wait for the response from the callee, this statement will lead to the wrong dependency graph. In contrast, our proposed solution does not make any parametric assumption nor requires any domain knowledge to find the root cause.

Checking dependence between variables. One straightforward way to check if a service is the root cause or not is to observe how much metrics of that service were affected after the failure. This is the strategy followed by ϵ-Diagnosis [43]. It uses a modified form of the coefficient of variation (COV) to check if a metric changed significantly during the failure. The critical problem with pairwise distance measures such as COV is that these measures do not condition on other variables which leads to a high false positive rate.

A common way to overcome the limitation of pairwise measures is to use conditional independence (CI). To this extent, most existing works use some version of the PC algorithm [37] to construct the dependency graph between the services [8, 53, 27, 14, 32]. There are two major problems with the PC algorithm. First, the PC algorithm only works with observational data and is therefore unable to use the information presented in the interventional data. This can lead to incomplete causal graphs (with observational data, we might only be able to find the correlation between two variables but with interventional data, we can infer the cause and effect). Second, the PC algorithm tries to learn the complete underlying causal graph which requires a higher number of conditional independence tests to be executed therefore making it impractical for applications with a large number of services.

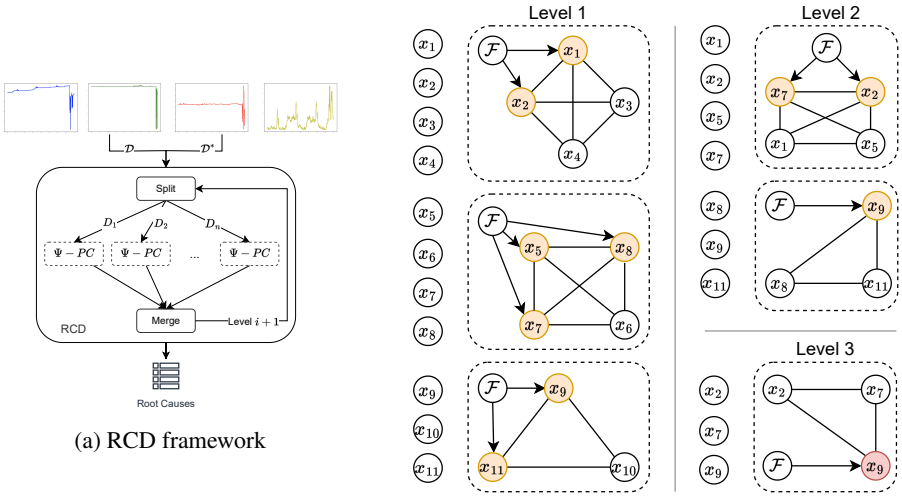
However, our method is capable of learning the causal structure by systematically leveraging the distributional invariance present not only within the observational data but also across the observational and interventional datasets. This has been proven to be strictly more informative than just using the observational dataset [38, 19]. Moreover, our proposed approach tries to find the root cause without learning the whole causal graph, therefore, reducing the number of CI tests and making PC practical for large-scale cloud applications.

4 Hierarchical Learning for RCA

We make the key observation that in a cluster of microservices, a failure can be thought of as performing a soft intervention on the failing node by changing its generative mechanism. Consider the latency of a microservice as a random variable L in a causal graph. During the normal state, it follows the probability distribution $P(L|Pa_L)$, where Pa_L is its parents in the graph. However, because of a programming bug or a configuration change, the probability distribution of the latency changes to $P' \neq P$. Such an effect (failure) can be considered a soft intervention as the affected node remains connected to its parents, but its conditional distribution changes. Accordingly, we define a variable R as a root-cause variable if $P(R|Pa_R)$ varies from normal mode of operation to the anomalous mode of operation. The key benefit of modeling the failure as an intervention is that it enables us to translate the problem of finding the root cause of failure into finding the interventional target. This view allows us to leverage the recent developments in the causal discovery literature that can identify intervention targets.

Learning Interventional Targets. We picked Ψ-FCI [19] which proposes the state-of-the-art sound and complete algorithm to learn the causal graph as well as the interventional targets from the observational and soft-interventional data. Ψ-FCI considers the case where the latent confounders are present in the data. Accordingly it uses a modified version of the FCI algorithm [47] to learn the causal graph. However, in our work we assume no confounder variables and therefore running FCI is not necessary. Accordingly, we designed a modified version of Ψ-FCI, named Ψ-PC, which internally uses PC [47], to find the interventional target considering no latent confounders .

Ψ-FCI, and consequently Ψ-PC introduces an extra node called the F-NODE to represent the effect of an intervention on the system. Such F-NODE's have been used for this purpose in Pearl's work [37] as a tool to prove some of the do-calculus rules. They were also used for causal discovery in several other studies, such as [33, 56]. The utility of such a representation is that one can identify distributional invariances of the form $P_N(X|Pa_X) = P_A(X|Pa_X)$, where P_N and P_A are the distributions under the normal mode of operation and anomalous mode of operation, respectively.



(a) RCD framework

(b) An execution of RCD with 11 nodes and $\gamma = 4$

Figure 1: The hierarchical and localized learning algorithm of RCD. It follows the divide-and-conquer approach by first splitting the dataset into small subsets to find interventional targets from each subset by running Ψ-PC (dashed boxes). In the merge phase, RCD combines the candidate root causes of all the subsets and performs the same steps recursively with the new set of variables. It stops when the set of candidate root causes cannot be trimmed further. Figure 1b shows an execution of RCD with 11 nodes where the orange nodes are potential root causes that are carried to the next level for further processing and the red node (x_9) is the eventual root cause. Note that RCD only estimates the neighborhood of F-NODE and leaves the rest of the graph untouched therefore reducing the number of CI tests significantly.

This can be done through conditional independence tests involving the F-NODE systematically by leveraging FCI or PC algorithms on the graph involving the F-NODE. Specifically, let us introduce the probability distribution P^* as $P^*(V|F=0) = P_N(V)$ and $P^*(V|F=1) = P_A(V)$, where V is the set of observed variables and F is the F-NODE. Then the invariance $P_N(X|Pa_X) = P_A(X|Pa_X)$ corresponds to the conditional independence statement $X \perp\!\!\!\perp F|Pa_X$, which can be tested on P^*. Samples from P^* can be obtained by simply concatenating normal and anomalous datasets with an F node that takes the value of 0 for normal and the value of 1 for the anomalous mode of operation. Running the PC algorithm [47] on this combined data with the domain knowledge that F-NODE cannot have any incoming edges allows us to discover all and only the root cause nodes, i.e., R such that $P_N(R|Pa_R) \neq P_A(R|Pa_R)$ as the remaining neighbors of F-NODE. The soundness and completeness of Ψ-PC follow from [19]. The detailed algorithm is presented in the appendix (Algorithm 2).

Hierarchical Learning. We propose a hierarchical learning algorithm called RCD that uses Ψ-PC as a tool to quickly find the interventional target. Figure 1a provides an overview of the hierarchical learning algorithm. Our key insight is that rather than finding the interventional target using all variables, we can divide the data into small subsets of variables, and find the interventional targets in every subset using Ψ-PC. The output of Ψ-PC is candidate interventional targets in each subset. With small sets, we reduce the number of CI tests and hence the run time. We use parameter γ to control the subset size. Note that partitioning the variables into subsets could lead to the descendants of the root cause and the actual root cause falling into different subsets. After running Ψ-PC on a subset, F-NODE will either point to the actual root cause if it exists in that subset or any descendant of the root cause that is not separable by conditioning on the nodes in that subset.

After running Ψ-PC on all the subsets, we take the union of the interventional targets from all subsets as potential root causes. Specifically, these are the variables that change their distribution from normal to the anomalous dataset, and whose change cannot be controlled for using the variables in that subset. However, it is possible that had we conditioned on variables in the other subsets, they would be conditionally independent of the F-NODE, and hence be removed from its neighborhood. In other words, the union of interventional targets will have false positive root causes. Accordingly, we can create new subsets using only the nodes in the current neighborhood of F-NODE and repeat this

process. At the final level, we will have a single set of nodes and running Ψ-PC on these reveals the actual root causes. Complete pseudo-code of the algorithm is given in Algorithm 1.

Algorithm 1 Root Cause Discovery Algorithm (RCD)

Input: Normal dataset $\mathcal{D}$, anomalous dataset $\mathcal{D}^*$, γ, Ψ-PC (.) [19], k : Max. no. of root causes
Output: A list of root causes $\mathcal{U}$.

procedure RCD($\mathcal{D}, \mathcal{D}^*, \gamma$)
  $\mathcal{U} \leftarrow$ Set of variables of $\mathcal{D}, \mathcal{D}^*$.
  while $|\mathcal{U}| > k$ **do**
    $\mathcal{S} \leftarrow$ A random partitioning of $|\mathcal{U}|$ into subsets of size γ.
    $R \leftarrow \emptyset$
    for all $S \in \mathcal{S}$ **do**
      $G \leftarrow \Psi\text{-PC}(\mathcal{D}[S], \mathcal{D}^*[S])$ # Ψ-PC constructs a graph on $S \cup \{\text{F-NODE}\}$.
      $R \leftarrow R \cup Ne_G(\text{F-NODE})$ # Extract neighbors of F-NODE.
    $\mathcal{U} \leftarrow R$.
return $\mathcal{U}$

Theorem 1 and its proof in the appendix states that RCD is sound for detecting the root causes.

Theorem 1 *Given access to a perfect conditional independence oracle, and under the causal sufficiency, and the extended faithfulness*[1] *assumptions Algorithm 1 returns the true root cause variables.*

Localized Learning. In the previous section, we discussed using RCD to find the root cause of the failure using Ψ-PC in a hierarchical fashion . Note that Ψ-PC focuses on learning not only the interventional targets but also the underlying causal graph. This is useful when some downstream task requires full causal structure [18, 57]. The time complexity of such an algorithm depends on the number of CI tests that need to be executed, which in general is exponential in the number of nodes for non-sparse graphs. By decoupling learning the causal graph from finding the interventional targets, we can improve run time by significantly reducing the number of CI tests.

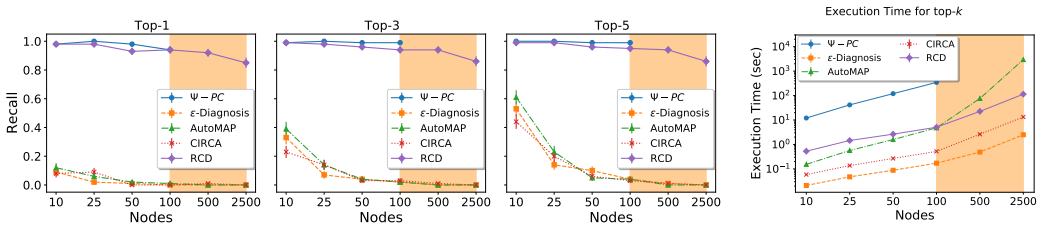
In our case, to find the root cause of the failure, we only need to learn the interventional target which translates to only learning the immediate neighborhood of the F-NODE. This key insight allows us to propose a localized learning algorithm that focuses on only learning the neighborhood of the F-NODE. The localized learning algorithm only runs the minimal set of CI tests needed to remove non-root cause nodes from the neighborhood of F-NODE. We can accomplish this by modifying Ψ-PC such that it only execute the CI test where one of the variables under consideration is F-NODE and therefore the rest of the graph will stay complete . Figure 1b shows an example of RCD with both hierarchical and localized algorithms. The localized version of the algorithm is used in our experiments, presented in the next section.

5 Evaluation

We conduct extensive evaluation of our proposed solution to answer the following questions: (1) *How effective is RCD for finding the interventional target?*, (2) *How quickly can RCD find the interventional target?* We report more detailed experiments in the appendix.

Implementation and Testing Setup. We implemented a modified version of RCD (Algorithm 1) for finite number of samples that generates a list of top-k potential root causes based on the p-values of CI tests. The complete algorithm with implementation details are provided in the Appendix. We implemented Ψ-PC in Python using the causal-learn package[2]. We used the Chi-squared test to check the independence between two variables. To generate synthetic data, we used pyAgrum[3] with a randomly generated DAG to draw samples for the normal and anomalous dataset. For all experiments, we set γ to 5 for RCD in all our experiments unless specified otherwise. Finally, to get statistically significant results, we ran all the experiments 100 times and plotted the average.

[1] The distributional invariances across datasets cannot be coincidental, please see [19] for a formal definition.
[2] github.com/cmu-phil/causal-learn
[3] pyagrum.readthedocs.io/en/1.0.0/



(a) Top-k recall

(b) Execution time

Figure 2: The left figure plots the top-k recall and the right one shows the execution time of different fault localization algorithms. RCD achieves similar top-k recall as Ψ-PC and it outperforms the other baselines because they either require domain knowledge, human intervention, or rely only on lower-order tests such as cross covariance. The execution time required for RCD increases slower with the number of nodes than Ψ-PC (note the logarithmic scale on the Y-axis), which does not terminate in a reasonable time for 500 and more nodes. AutoMAP performs better than Ψ-PC but still incur a high execution time because it internally uses PC. ϵ-Diagnosis achieves the best execution time but its usefulness is limited as its recall drops significantly when the number of node increases.

Baseline and Evaluation Metrics. We compare our solution to the following baseline methods;

- ϵ-Diagnosis [43]: The authors used the coefficient of variation (COV) on the normal and anomalous data to find microservices that changed significantly after the failure. The comparison with ϵ-Diagnosis will show the importance of our use of the CI test.
- AutoMAP [27]: AutoMAP constructs a weighted causal graph using a modified version of PC algorithm where the edge weight between the two microservices signifies the (in)dependence of a performance metric. It concludes by traversing nodes based on a random walk algorithm and finding the root cause from the weighted causal graph using the correlation metric.
- CIRCA [22]: CIRCA uses service call graph to construct a causal graph between the metrics with a set of static and domain-specific rules. It runs a regression hypothesis test on the anomalous data to find the deviation of a variable from its expected normal distribution after the failure. The comparison with CIRCA shows that tools that make parametric assumptions are applicable only under strict constraints and hence fall short in real-world applications.
- Ψ-PC: The comparison with Ψ-PC will highlight that learning the complete causal graph is not necessary for finding the root cause. Furthermore, it will also establish the time gain RCD gets from localizing learning.
- RCD: The proposed hierarchical solution with localized learning.

Contrary to RCD, CIRCA requires a call graph to build a causal structure between the microservices. For this purpose, we used the output of PC as the call graph for CIRCA. Two quantitative metrics are generally used to evaluate the performance of different algorithms related to root cause detection: execution time and recall at top-k [27, 54, 32, 53, 22]. In our case, recall at top-k is the probability of getting the correct root cause from the top k outputs of the algorithm.

Finding Root Cause Our first experiment compares the detectability of different RCA algorithms. We ran this experiment by controlling the number of nodes in the DAG. The max in-degree of all the DAGs are set to 3 and the total number of states for every node is 6. Furthermore, for every experiment, we draw 10K samples for the normal and anomalous states of the system.

Figure 2a shows the top-k recall with $k = 1/3/5$. In general, we see that RCD can identify the interventional target (root cause) with very high recall. More specifically, it achieves 98% recall at top-1. In terms of other algorithms, Ψ-PC performs similarly to RCD (but cannot run beyond 100 node). This is because RCD internally uses a modified version of Ψ-PC to learn the interventional target, but solves the scalability problem of Ψ-PC. On the flip side, recall of AutoMAP for $k = 5$ is just above 50% when there are only 10 nodes and it drops to 0 when the number of nodes increases. One of the key reasons for this result is that the accuracy of AutoMAP depends on the historical data about the failures. Hence, for the initial rounds of the execution (i.e., without expert human input), AutoMAP performs worse as it only relies on the correlation between the variables. ϵ-Diagnosis

performs much worse than RCD because of its use of COV which does not take into account other variables when deciding the independence between two variables. ϵ-Diagnosis performs worse with a larger number of nodes as there is more room to miss the root cause and classify a wrong microservice to be the root cause simply because that metric shows a bigger change. Finally, we observe that CIRCA is unable to find the root cause with higher number of nodes. This is because our data generation model does not follow any specific parametric distribution which makes it difficult to find the root cause using tools that rely on regression-based techniques.

Execution Time. Most microservice-based cloud applications are latency sensitive thus any tool that claims to find the root cause of the failure has to minimize the execution time. To measure the execution time of different algorithms, we conducted experiments on synthetic data with a varying number of variables and drew 10K samples for both datasets (normal and anomalous). We measured the time it takes to find the top-k potential root causes where $k = 1/3/5$. Figure 2b illustrates the results.

The first thing to note here is that RCD outperforms all the baselines. On average, for 500 nodes, it can detect the actual root cause of the failure in the top 5 candidates in just above 22 seconds. Secondly, the execution time of Ψ-PC grows exponentially as the number of nodes increases. For just 500 nodes, single execution of Ψ-PC was taking more than 150 minutes. Here we can observe the benefits of using a hierarchical and localized approach to learn the root cause. Because of the time constraint, we only ran Ψ-PC for a maximum of 100 nodes.

Considering other baselines, we see that AutoMAP still incurs a high delay because of its dependence on PC algorithm which tries to learn the whole causal graph. Furthermore, the execution time of ϵ-Diagnosis almost stays constant. The reason for this is that it only considers variables in a pairwise fashion, thus, the number of tests for COV grows only quadratically with the number of nodes. The consequence of only considering the variables in a pairwise fashion is that it leads to a higher number of false positives which significantly affects the top-k recall. Similarly, inference from CRICA is fast because it does not build a causal graph but requires the call graph as an input[4].

The key takeaway from our experiments is that RCD provides almost identical top-k recall as compared to Ψ-PC while reducing the execution time significantly. RCD gains this benefit by (1) exploring the data in a hierarchical fashion and (2) by only learning the neighborhood of F-NODE rather than learning the complete underlying causal graph.

6 Case Study

Sock-shop. To test the applicability and quantify the performance gain of different root cause discovery algorithms, we setup a test-bed using a real microservice-based application. We use Sock-shop [44], a replica of an online application that sells socks. It consists of 13 microservices written in different technologies and each microservice is deployed on its separate VM/container. The communication between the microservices happens through API requests over HTTP. Moreover, all these microservices provide a large amount of statistical information in terms of different metrics (for instance, CPU and memory utilization, latency, and the number of errors). We also developed an independent workload generator using Locust [5] to send the traffic to the Sock-shop application. To mimic a real application, we followed sinusoidal distribution with a mean of 50 to draw the number of users for every second. Every user first signs up on the system, browse a few items, and finally orders one. This whole process sends multiple requests to all the microservices.

There are two very common failures in cloud computing applications: CPU hog and memory leak [8, 28, 27]. We inject these two types of failures in microservices by running `stress-ng` [6]. We accomplish this by modifying the Docker images of all the microservices to install `stress-ng` and then running it on the container of the targeted microservice. With this setup, we injected failures in all of the major microservices of Sock-shop (carts, catalogue, orders, payment, and user). Out of the 13 microservices, we selected these because they were the critical user-facing services. Any change to their performance cause the other microservices to get affected as well, thus making it difficult to detect the root cause with a simple threshold-based scheme.

[4] For CIRCA, we did not include the time for running PC to construct the call graph

[5] locust.io/

[6] wiki.ubuntu.com/Kernel/Reference/stress-ng

Table 1: Top-k recall of different algorithms for detecting the root cause of data from the Sock-shop application. Ψ-PC suffers mainly because of the limited number of samples whereas RCD can achieve reasonable recall due to localized search. AutoMAP performs worse because of its reliance on the historical data.

		CPU hog				Memory leak			
		Ψ-PC	AMAP	ϵ-Diag.	RCD	Ψ-PC	AMAP	ϵ-Diag.	RCD
Top-1	Carts	0.6	0.05	0.2	**0.69**	0.0	0.14	0.4	**0.6**
	Catalogue	0.0	**0.22**	0.2	0.17	0.0	0.0	**0.2**	0.11
	Orders	0.0	0.01	0.2	**0.29**	0.0	0.26	0.2	**0.45**
	Payment	0.0	0.07	**0.6**	0.17	0.0	0.01	**0.4**	0.17
	User	0.0	0.29	0.2	**0.46**	0.2	0.08	0.2	**0.71**
Top-3	Carts	0.6	0.23	0.4	**0.86**	0.0	0.2	0.4	**0.88**
	Catalogue	0.0	**0.22**	0.2	**0.22**	0.0	0.0	0.2	**0.33**
	Orders	0.0	0.36	0.2	**0.66**	0.0	0.37	0.2	**0.56**
	Payment	0.0	0.07	**0.6**	0.4	0.0	0.05	**0.8**	0.34
	User	0.0	0.53	0.2	**0.66**	0.2	0.27	0.2	**0.84**
Top-5	Carts	0.6	0.24	0.4	**0.86**	0.0	0.25	0.4	**0.88**
	Catalogue	0.0	**0.23**	0.2	0.22	0.0	0.0	0.2	**0.33**
	Orders	0.0	0.51	0.2	**0.66**	0.0	0.49	0.2	**0.56**
	Payment	0.0	0.07	**0.6**	0.40	0.0	0.11	**0.8**	0.34
	User	0.0	0.55	0.2	**0.66**	0.2	0.3	0.2	**0.84**

We ran the Sock-shop application for 5 minutes to collect the data for the normal state of the system. After that, we injected the failure and executed the application for 5 more minutes to collect the anomalous dataset. We repeated the experiment 5 times for two different types of failures (CPU hog and memory leak) and in total, we gathered 50 datasets. To measure the performance of the root cause detection algorithm, we ran the algorithm 100 times on every dataset and quantified its top-k recall. In this context, the top-k recall can be interpreted as the probability that the algorithm can find the actual root cause in a list of top-k potential root causes.

Table 1 shows the top-k recall of different algorithms for detecting the root cause of the data collected from Sock-shop. We report the execution time in table Table 3 in the appendix. RCD performs better than the baselines in general. Considering baselines, Ψ-PC performs poorly for most of the microservices. The main reason for this is the limited number of samples for the normal dataset, which makes Ψ-PC unable to learn the required causal structure. In a few cases, we observe that ϵ-Diagnosis performs better than RCD because of the fewer number of services available in Sock-shop. However as shown earlier from our experiments with synthetic data, ϵ-Diagnosis does not scale well with large number of nodes.

Real Data. We further validate RCD on a set of real-world datasets collected from a part of a production-based microservice system hosted on AWS cloud-native system. The data was collected from Grafana, a monitoring tool for cloud applications. The system consists of 25 different microservices. A total of 150 metrics were collected which broadly cover all the monitoring interfaces of the services. The type of metrics ranged from throughput, memory utilization, service latency to I/O read-writes and system load, *i.e.*, covering the golden signals [4].

We collected data for three distinct outages that occurred in the last 6 months in the system. For every failure, a team of Site Reliability Engineers (SREs) report the time when the outage occurred and when it was fixed. Accordingly, we collected anomalous data for the duration of the failure whereas the normal data contains the metric information for 2 days before the failure. The summary of the outages and the result from RCD and ϵ-Diagnosis are shown in Table 2. Ψ-PC was unable to complete its execution within an hour and hence we exclude it. AutoMAP requires a front-end service and needs the same set of metrics to be available for all the services which were not true in our case. Finally, CIRCA requires the metrics to be classified into the metric categories and a call graph which were unavailable to us and hence we exclude it as well. On the flip side, RCD does not require any such information or domain knowledge. Next, we discuss the results of every failure in detail.

Outage $\mathcal{A}$. According to the incident report, the outage lasted for over an hour. It occurred during a planned database maintenance that involved shunning and rebuilding database of every node individually from a cluster. However, during the rebuilding phase, AWS autoscaling was not able to

Table 2: The summary and top-5 recall of ϵ-Diagnosis and RCD on data collected for three outages from a production-based cloud application. The length of the vector represents the number of services that were flagged as root cause and the individual number shows the number of metrics belonging to a particular service. The green color illustrates that algorithm was able to correctly detect the root cause whereas the red color shows the algorithm could not find the root cause.

			Rank of Services from top-5		Time (sec)	
Outage	Metrics	Duration (min)	ϵ-Diagnosis	RCD	ϵ-Diagnosis	RCD
$\mathcal{A}$	137	65	[**1**,1,1,1,1]	[**2**,1,1,1]	0.145	112
$\mathcal{B}$	147	72	[**1**,1,1,1,1]	[**3**,1,1]	0.186	239.8
$\mathcal{C}$	150	210	[**2**,1,1,1]	[**4**,1]	0.146	22.57

provision a similar instance in the same zone due to capacity issues. This resulted in an imbalance in autoscaling which trickled down as a series of faults. On running RCD on this dataset, it found the memory footprint of different tiers of the database as the top-2 root causes, while the resource utilization of the database for the message queuing system was flagged within the top-7 root causes.

Outage $\mathcal{B}$. During this outage, which lasted for about an hour and 12 minutes, the service instances located at a specific region were unavailable affecting several customers. The system failed due to a fault in the event consumer queue which got stuck in one of the microservice. The consequence was longer database query time and increased memory size in the application server. Executing RCD with the outage data found the heap size and the system load of the event queue to be the culprit among the top-3 root cause. However, the top-1 root cause, which was the hit ratio in the Memcached database was not among the faulty services. We believe, there might have been some latent variable between the failed service and the Memcached database that was affecting both services. We leave the problem of finding root causes with latent variables for future work.

Outage $\mathcal{C}$. In another outage that lasted for around 3.5 hours, alerts regarding high error rates in a microservice (donated as $\mathcal{M}$ here for simplicity) were fired. The users were facing errors while accessing $\mathcal{M}$ however, the team of SREs found no issues with the services. Later it was found that a faulty update in one of the AWS components has caused that component to fail. That component was being used by $\mathcal{M}$ and therefore after the update $\mathcal{M}$ was unable to process incoming requests. Note that in this outage, the system was not collecting metrics information about faulty AWS components because it was out of the boundary of the system. However, running RCD on the metrics data available, it found the disk space, error rate, and heap size of $\mathcal{M}$ to be the root cause metrics. Even though the faulty component was outside the system, RCD was able to find the directly affected service as the root cause. This type of information is immensely useful for SREs to narrow down the problem during system diagnosis.

7 Conclusion

We proposed a novel root cause analysis algorithm that systematically leverages the distributional invariances across normal and anomalous datasets, taking inspiration from the recent causal discovery algorithms. Our tailored algorithm sidesteps learning the full causal graph and rather focuses on only the root causes. This provides not only significant benefits in runtime, but also in the number of required anomalous samples.

8 Acknowledgement

We appreciate the anonymous reviewers for their valuable and constructive feedback that greatly improved the manuscript. We would also like to thank Ahaan Dabholkar for assistance with running and automating the experiments.

References

[1] Andrew Arnold, Yan Liu, and Naoki Abe. Temporal causal modeling with graphical granger methods. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 66–75, 2007.

[2] Elias Bareinboim and Judea Pearl. Causal inference and the data-fusion problem. *Proceedings of the National Academy of Sciences*, 113(27):7345–7352, 2016.

[3] Yogesh D Barve, Shashank Shekhar, Ajay Chhokra, Shweta Khare, Anirban Bhattacharjee, Zhuangwei Kang, Hongyang Sun, and Aniruddha Gokhale. Fecbench: A holistic interference-aware approach for application performance modeling. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 211–221. IEEE, 2019.

[4] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Inc., 1st edition, 2016.

[5] Kenneth A Bollen. *Structural equations with latent variables*, volume 210. John Wiley & Sons, 1989.

[6] Philippe Brouillard, Sébastien Lachapelle, Alexandre Lacoste, Simon Lacoste-Julien, and Alexandre Drouin. Differentiable causal discovery from interventional data. *Advances in Neural Information Processing Systems*, 33:21865–21877, 2020.

[7] Mike Y Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings International Conference on Dependable Systems and Networks*, pages 595–604. IEEE, 2002.

[8] Pengfei Chen, Yong Qi, Pengfei Zheng, and Di Hou. Causeinfer: Automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*, pages 1887–1895. IEEE, 2014.

[9] David Maxwell Chickering. Optimal structure identification with greedy search. *Journal of machine learning research*, 3(Nov):507–554, 2002.

[10] Scott Emmons Coburn Watson and Brendan Gregg. A microscope on microservices netflixtechblog.com/a-microscope-on-microservices-923b906103f4/. *Netflix Technology Blog*, 2022.

[11] Daniel Eaton and Kevin Murphy. Exact bayesian structure learning from uncertain interventions. In *Artificial intelligence and statistics*, pages 107–114. PMLR, 2007.

[12] Rodrigo Fonseca, George Porter, Randy H Katz, and Scott Shenker. {X-Trace}: A pervasive network tracing framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*, 2007.

[13] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *2009 ninth IEEE international conference on data mining*, pages 149–158. IEEE, 2009.

[14] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. Sage: practical and scalable ml-driven performance debugging in microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 135–151, 2021.

[15] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 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*, pages 19–33, 2019.

[16] Clark Glymour, Kun Zhang, and Peter Spirtes. Review of causal discovery methods based on graphical models. *Frontiers in genetics*, 10:524, 2019.

[17] Xiaofeng Guo, Xin Peng, Hanzhang Wang, Wanxue Li, Huai Jiang, Dan Ding, Tao Xie, and Liangfei Su. Graph-based trace analysis for microservice architecture understanding and problem diagnosis. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1387–1397, 2020.

[18] Paul Hünermund and Elias Bareinboim. Causal inference and data fusion in econometrics. *arXiv preprint arXiv:1912.09104*, 2019.

[19] Amin Jaber, Murat Kocaoglu, Karthikeyan Shanmugam, and Elias Bareinboim. Causal discovery from soft interventions with unknown targets: Characterization and learning. *Advances in neural information processing systems*, 33:9551–9561, 2020.

[20] Myunghwan Kim, Roshan Sumbaly, and Sam Shah. Root cause detection in a service-oriented architecture. *ACM SIGMETRICS Performance Evaluation Review*, 41(1):93–104, 2013.

[21] Murat Kocaoglu, Amin Jaber, Karthikeyan Shanmugam, and Elias Bareinboim. Characterization and learning of causal graphs with latent variables from soft interventions. *Advances in Neural Information Processing Systems*, 32, 2019.

[22] Mingjie Li, Zeyan Li, Kanglin Yin, Xiaohui Nie, Wenchi Zhang, Kaixin Sui, and Dan Pei. Causal inference-based root cause analysis for online service systems with intervention recognition. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 3230–3240, 2022.

[23] Zeyan Li, Junjie Chen, Rui Jiao, Nengwen Zhao, Zhijun Wang, Shuwei Zhang, Yanjun Wu, Long Jiang, Leiqin Yan, Zikai Wang, et al. Practical root cause localization for microservice systems via trace analysis. In *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*, pages 1–10. IEEE, 2021.

[24] JinJin Lin, Pengfei Chen, and Zibin Zheng. Microscope: Pinpoint performance issues with causal graphs in micro-service environments. In *International Conference on Service-Oriented Computing*, pages 3–20. Springer, 2018.

[25] Haifeng Liu, Jinjun Zhang, Huasong Shan, Min Li, Yuan Chen, Xiaofeng He, and Xiaowei Li. Jcallgraph: Tracing microservices in very large scale container cloud platforms. In *International Conference on Cloud Computing*, pages 287–302. Springer, 2019.

[26] Ping Liu, Haowen Xu, Qianyu Ouyang, Rui Jiao, Zhekang Chen, Shenglin Zhang, Jiahai Yang, Linlin Mo, Jice Zeng, Wenman Xue, et al. Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pages 48–58. IEEE, 2020.

[27] Meng Ma, Jingmin Xu, Yuan Wang, Pengfei Chen, Zonghua Zhang, and Ping Wang. Automap: Diagnose your microservice-based web applications automatically. In *Proceedings of The Web Conference 2020*, pages 246–258, 2020.

[28] Leonardo Mariani, Cristina Monni, Mauro Pezzé, Oliviero Riganelli, and Rui Xin. Localizing faults in cloud systems. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 262–273. IEEE, 2018.

[29] Dimosthenis Masouros, Sotirios Xydis, and Dimitrios Soudris. Rusty: Runtime interference-aware predictive monitoring for modern multi-tenant systems. *IEEE Transactions on Parallel and Distributed Systems*, 32(1):184–198, 2020.

[30] Chris Meek. *Complete orientation rules for patterns*. Carnegie Mellon [Department of Philosophy], 1995.

[31] Christopher Meek. Causal inference and causal explanation with background knowledge. *arXiv preprint arXiv:1302.4972*, 2013.

[32] Yuan Meng, Shenglin Zhang, Yongqian Sun, Ruru Zhang, Zhilong Hu, Yiyin Zhang, Chenyang Jia, Zhaogang Wang, and Dan Pei. Localizing failure root causes in a microservice through causality inference. In *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2020.

[33] Joris M Mooij, Sara Magliacane, and Tom Claassen. Joint causal inference from multiple contexts. *Journal of machine learning research*, 2020.

[34] Raha Moraffah, Mansooreh Karami, Ruocheng Guo, Adrienne Raglin, and Huan Liu. Causal interpretability for machine learning-problems, methods and evaluation. *ACM SIGKDD Explorations Newsletter*, 22(1):18–33, 2020.

[35] Sam Newman. *Building microservices*. " O'Reilly Media, Inc.", 2021.

[36] Rainer Opgen-Rhein and Korbinian Strimmer. From correlation to causation networks: a simple approximate learning algorithm and its application to high-dimensional plant gene expression data. *BMC systems biology*, 1(1):1–10, 2007.

[37] Judea Pearl. Causal diagrams for empirical research. *Biometrika*, 82(4):669–688, 1995.

[38] Judea Pearl. *Causality*. Cambridge university press, 2009.

[39] Judea Pearl. The seven tools of causal inference, with reflections on machine learning. *Communications of the ACM*, 62(3):54–60, 2019.

[40] Mohammad Farhad Peerally, Susan Carr, Justin Waring, and Mary Dixon-Woods. The problem with root cause analysis. *BMJ quality & safety*, 26(5):417–422, 2017.

[41] Juan Qiu, Qingfeng Du, Kanglin Yin, Shuang-Li Zhang, and Chongshu Qian. A causality mining and knowledge graph based method of root cause diagnosis for performance anomaly in cloud applications. *Applied Sciences*, 10(6):2166, 2020.

[42] Joseph Ramsey, Madelyn Glymour, Ruben Sanchez-Romero, and Clark Glymour. A million variables and more: the fast greedy equivalence search algorithm for learning high-dimensional graphical causal models, with an application to functional magnetic resonance images. *International journal of data science and analytics*, 3(2):121–129, 2017.

[43] Huasong Shan, Yuan Chen, Haifeng Liu, Yunpeng Zhang, Xiao Xiao, Xiaofeng He, Min Li, and Wei Ding. ?-diagnosis: Unsupervised and real-time diagnosis of small-window long-tail latency in large-scale microservice platforms. In *The World Wide Web Conference*, pages 3215–3222, 2019.

[44] Sock-shop - a microservice demo application github.com/microservices-demo/microservices-demo. 2022.

[45] Jacopo Soldani and Antonio Brogi. Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey. *ACM Comput. Surv.*, 55(3), feb 2022.

[46] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, 146:215–232, 2018.

[47] Peter Spirtes, Clark N Glymour, Richard Scheines, and David Heckerman. *Causation, prediction, and search*. MIT press, 2000.

[48] Jörg Thalheim, Antonio Rodrigues, Istemi Ekin Akkus, Pramod Bhatotia, Ruichuan Chen, Bimal Viswanath, Lei Jiao, and Christof Fetzer. Sieve: Actionable insights from monitored metrics in distributed systems. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, pages 14–27, 2017.

[49] Sofia Triantafillou and Ioannis Tsamardinos. Constraint-based causal discovery from multiple interventions over overlapping variable sets. *The Journal of Machine Learning Research*, 16(1):2147–2205, 2015.

[50] Burak Varici, Karthikeyan Shanmugam, Prasanna Sattigeri, and Ali Tajer. Scalable intervention target estimation in linear models. *Advances in Neural Information Processing Systems*, 34:1494–1505, 2021.

[51] Hanzhang Wang, Phuong Nguyen, Jun Li, Selcuk Kopru, Gene Zhang, Sanjeev Katariya, and Sami Ben-Romdhane. Grano: Interactive graph-based root cause analysis for cloud-native distributed data platform. *Proceedings of the VLDB Endowment*, 12(12):1942–1945, 2019.

[52] Hanzhang Wang, Zhengkai Wu, Huai Jiang, Yichao Huang, Jiamu Wang, Selcuk Kopru, and Tao Xie. Groot: An event-graph-based approach for root cause analysis in industrial settings. *arXiv preprint arXiv:2108.00344*, 2021.

[53] Ping Wang, Jingmin Xu, Meng Ma, Weilan Lin, Disheng Pan, Yuan Wang, and Pengfei Chen. Cloudranger: Root cause identification for cloud native systems. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 492–502. IEEE, 2018.

[54] Li Wu, Johan Tordsson, Jasmin Bogatinovski, Erik Elmroth, and Odej Kao. Microdiag: Fine-grained performance diagnosis for microservice systems. In *2021 IEEE/ACM International Workshop on Cloud Intelligence (CloudIntelligence)*, pages 31–36. IEEE, 2021.

[55] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 117–132, 2009.

[56] Karren Yang, Abigail Katcoff, and Caroline Uhler. Characterizing and learning equivalence classes of causal dags under interventions. In *International Conference on Machine Learning*, pages 5541–5550. PMLR, 2018.

[57] Changwon Yoo, Vesteinn Thorsson, and Gregory F Cooper. Discovery of causal relationships in a gene-regulation pathway from a mixture of experimental and observational dna microarray data. In *Biocomputing 2002*, pages 498–509. World Scientific, 2001.

[58] Guangba Yu, Pengfei Chen, Hongyang Chen, Zijie Guan, Zicheng Huang, Linxiao Jing, Tianjun Weng, Xinmeng Sun, and Xiaoyun Li. Microrank: End-to-end latency issue localization with extended spectrum analysis in microservice environments. In *Proceedings of the Web Conference 2021*, pages 3087–3098, 2021.

[59] Keli Zhang, Marcus Kalander, Min Zhou, Xi Zhang, and Junjian Ye. An influence-based approach for root cause alarm discovery in telecom networks. In *International Conference on Service-Oriented Computing*, pages 124–136. Springer, 2020.

[60] Kun Zhang, Biwei Huang, Jiji Zhang, Clark Glymour, and Bernhard Schölkopf. Causal discovery from nonstationary/heterogeneous data: Skeleton estimation and orientation determination. In *IJCAI: Proceedings of the Conference*, volume 2017, page 1347. NIH Public Access, 2017.

[61] Laiping Zhao, Yanan Yang, Kaixuan Zhang, Xiaobo Zhou, Tie Qiu, Keqiu Li, and Yungang Bao. Rhythm: component-distinguishable workload deployment in datacenters. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–17, 2020.

[62] Nengwen Zhao, Honglin Wang, Zeyan Li, Xiao Peng, Gang Wang, Zhu Pan, Yong Wu, Zhen Feng, Xidao Wen, Wenchi Zhang, et al. An empirical investigation of practical log anomaly detection for online service systems. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1404–1415, 2021.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] Conclusion contains limitations of our work.
   (c) Did you discuss any potential negative societal impacts of your work? [N/A]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes]
   (b) Did you include complete proofs of all theoretical results? [Yes]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [No] We plan to share our code and the data in the future.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] Some of the error bars are tiny because of small variance.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [No]
   (c) Did you include any new assets either in the supplemental material or as a URL? [No]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

Appendix

This section discusses some details about root cause analysis and RCD in details.

Preliminaries

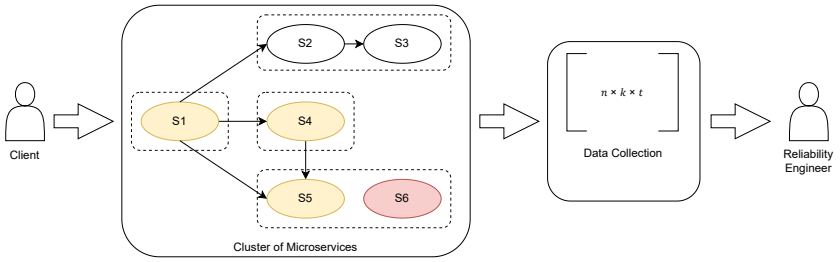


Figure 3: The workflow of a microservice-based system from the usage point of a client and a Site Reliability Engineer (SRE). Every oval is a microservice whereas every dashed rectangle is a host/VM that hosts multiple microservices. The edges between two microservices represent the call graph. Red-colored microservices signify a failure but the effect is observed on the yellow-colored microservices. Even with the topological information about the microservices, it will be difficult for a SRE to pinpoint the root cause just observing the failures at, say $s5$.

Microservice Architecture. Microservices are usually containerized applications that perform a small but specific set of tasks independently from other microservices. In a microservice-based application, a set of microservices work together to fulfill a client's request. For monitoring and debugging purposes, all of the microservices expose some kind of performance metrics. There are many tools that help developers instrument the microservices to expose useful information as well as collect, store and visualize all that data. Most of these tools scrap all the microservices periodically, store the metric information, and produce a time-series dataset. SREs use this time-series metric data to monitor the performance of the overall application as well as the health of individual microservices. Figure 3 depicts a general web application with multiple microservices running on different hosts.

Causality and Causal Graph. We define causality as a means to measure the dependence between two variables. One way to depict the causal relation between these variables is the causal graph. In the causal graph, every node represents a random variable whereas an edge between the two nodes shows the dependence between the variables. The conditional probability distribution of a node in the causal graph can be computed as

$$P(X_i|pa_i) \tag{1}$$

Here X_i represents the random variable i in the causal graph and pa_i is the set of parents. We can use 1 to compute the marginal probability distribution of a node by summing over the probability of all the values of pa_i.

$$P(X_i) = \sum_x P(X_i|x)P(x) \tag{2}$$

Moreover, the probability of the overall system with n random variables can be computed as

$$P(X_1, \ldots, X_n) = \prod_i P(X_i|pa_i) \tag{3}$$

We can use 1 as the conditional independence (CI) test to check the (in)dependence relationship between two variables. For instance, two variables X and Y are independent only if we can find a set $S \cap \{X, Y\} = \emptyset$ such that $P(X|Y, S) = P(X|S)$.

Formally, a causal graph can be defined as a DAG $G(V, E)$ where V is the set of n vertices, $V = \{v_1, v_2, ...v_n\}$, and $E \subseteq V \times V$ is the set of the directed and undirected edges. A directed edge $\vec{e} = \{v_1, v_2\}$ signifies that v_1 causes v_2. Here the edge $\vec{e}$ shows that we could not find a *separating set*, S, such that $P(v_1|v_2, S) = P(v_1|S)$.

Causal Discovery Causal discovery is a well-known problem in causal inference and has been studied thoroughly [38, 56, 6, 2]. It has applications in social sciences, bioinformatics, economics, and has recently found its way into machine learning as well [36, 5, 34, 39, 57, 18]. Broadly speaking

there are two type of algorithms for causal discovery, **Score-based causal discovery** [9, 42] and **Constraint-based causal discovery** [47, 16]. Score-based algorithms aim to find the causal structure by optimizing a properly defined score function. Among them, Greedy Equivalence Search (GES) [9] is a well-known two-phase procedure that directly searches over the space of equivalence classes.

The constraint-based causal discovery algorithms, on the other hand, start with a complete graph and remove the edges between a pair of variables if they are conditionally independent. To test the independence of two variables, denoted as $X \perp\!\!\!\perp Y$, it uses some well-established statistical conditional independence (CI) tests such as Fisher z-transformation or G-test for hypothesis testing. In contrast to score-based causal discovery algorithms, there have been several studies done for constraint-based algorithms in the presence of interventional data [21, 33, 60].

A renowned constraint-based causal discovery algorithm is the PC algorithm [47]. PC algorithm works in two phases. The first phase is where it tries to learn the skeleton graph (undirected graph) and in the second phase, it tries to orient the edges between the nodes. Starting with a completely connected graph, it picks a connected pair of variables; let us call them X and Y, and runs a CI test to check if the variables are dependent or not. If they are independent, $X \perp\!\!\!\perp Y$, it removes the edge between the two variables. Next, if two variables are still connected in the graph, it tries to find a separating set S, such that conditioning on that set makes the variables independent. To try to find the smallest S, after every iteration, it increases the cardinality of the separating set by 1.

The output of the first phase of the PC algorithm is an undirected skeleton graph, where an edge represents a dependence between the two connecting variables. The next step is to orient the edges of the skeleton graph. The PC algorithm achieves this with a set of rules [30].

From the causal inference literature, there are a plethora of studies that try to learn the underlying causal structure in the presence of observation and interventional data [49, 2, 6, 21, 56]. Furthermore, there are also a set of studies that try to identify the interventional target (the node where intervention happened) [33, 11, 60, 19].

Proof of Theorem 1.

Note that Ψ-PC concatenates two datasets with an F-NODE and runs PC algorithm on all the variables by treating F-NODE as another variable. Even though with more than two datasets this would create faithfulness violations among multiple F-NODE's as pointed out in [19, 33], no such issue arises when there are two datasets creating only a single F-NODE. Accordingly, Ψ-PC can avoid the careful treatment between F-NODE's that Ψ-FCI requires. Same for RCD which uses Ψ-PC.

We will use p_N to represent the probability distribution of variables under normal operating conditions (observational), and p_A to represent the one under anomaly (interventional). Let us introduce the probability distribution p^* that is defined as $p^*(V|F=0) = p_N(V)$ and $p^*(V|F=1) = p_A(V)$, where V is the set of observed variables and F is the F-NODE representing the effect of intervention.

First, note that if Ψ-PC was run on the full graph, it would output a graph where the F-NODE is adjacent to only the true root causes. This is because any variable S that is not a root cause can be separated from the F-NODE as $S \perp\!\!\!\perp F \,|Pa_S$, since $p_N(s|pa_s, F=0) = p_A(s|pa_s, F=1)$ and Pa_S is observable for all S due to the causal sufficiency assumption.

RCD algorithm runs Ψ-PC on a subset of variables in every level. F-NODE will then remain adjacent to those that are not separable from it by conditioning on any subset of the variables in that subset. Due to the extended faithfulness assumption, we have that $p_N(s|u, F=0) \neq p_A(s|u, F=1)$ for any root cause node S, and for any subset U of the observed variables, or equivalently $S \not\perp\!\!\!\perp F \,|U$. Therefore, root-causes cannot be separated using any subset of the observed variables. This establishes that, whichever subset the true root causes fall into, they will all remain adjacent to the F-NODE of that subset. This is true for all levels of the algorithm. This establishes the soundness of the algorithm, that at any level, including the final level, root-causes will remain adjacent to the F-NODE.

RCD however is not complete. We have the following simple example to demonstrate that the returned set might contain additional nodes that are not root causes. Consider the augmented graph $F \to R \to S, T \to R, T \to S$. Suppose we partition the nodes into subsets $\{\{T\}, \{R, S\}\}$. F-NODE will not be adjacent to T since $F \perp\!\!\!\perp T$. For the second subset, F-NODE will be adjacent to both R, S. This is because one needs to condition on both R and T in order to d-separate F-NODE from S.

However, T was removed in the first subset and is not included in the next stage. Hence at the end of the execution, RCD will be connected to both R and S even though the actual root cause is R.

Please note that one can easily resolve this issue by running one more stage at the end of RCD algorithm by including all the nodes $\{U : F \perp\!\!\!\perp U\}$. These are the nodes that are non-descendants of the F-NODE, which contains the parents of the root causes. Including them enables finding a valid separating set for any node that is not a root cause. This is because R, Pa_R blocks all the backdoor and frontdoor paths from the root causes that would d-connect F-NODE and any non root cause node.

In our experiments we only implemented the vanilla RCD which is sound but not complete as it was sufficient to achieve the recall of Ψ-PC. Furthermore, existence of false positives is not a problem for root cause analysis application as long as most of the variables are eliminated quickly as potential root causes in an automated manner.

Hierarchical Learning Algorithm

In this section, we provide a modified version of RCD when the number of samples are finite. Similar to Algorithm 1, the algorithm for finite number of samples is divided in two phases as well. In the first phase, we divide the data in small subsets of size γ and run Ψ-PC to learn the potential interventional targets. In the second phase, we take the union of all the potential interventional targets and narrow the list down to k interventional targets. Complete pseudocode of this algorithm is provided in Algorithm 3.

Algorithm 2 Ψ-PC: Algorithm for learning the causal graph and interventional targets from observational and interventional probability distributions [19]

Input: Observational distribution P_N, interventional distribution P_A, and set of variables V
Output: A causal graph on $\mathcal{G}$.

procedure Ψ-PC(P_N, P_A, V)
  $P^*(V|\text{F-NODE} = 0) \leftarrow P_N(V), P^*(V|\text{F-NODE} = 1) \leftarrow P_A(V)$ # Concatenate
  $P^*(\text{F-NODE} = 0) \leftarrow 0.5, P^*(\text{F-NODE} = 1) \leftarrow 0.5$ # Equal number of samples
  Phase-1: Learn Skeleton
  $\mathcal{G} \leftarrow$ a complete undirected graph over $V \cup \{\text{F-NODE}\}$
  for all every pair $X, Y \in V \cup \{\text{F-NODE}\}$ **do**
    for all $S \subseteq V \setminus \{X\}$ **do**
      if $P^*(X|Y, S) = P^*(X|S)$ **then**
        $SepSet(X, Y) \leftarrow S$ and remove the edge between X and Y from $\mathcal{G}$
  Phase-2: Edge Orientation
  $\mathcal{R}_0$: For any triplets $< X, Z, Y >$, such that $X \cap Y = \emptyset$, orient $X \rightarrow Z \leftarrow Y$ iff $Z \notin SepSet(X, Y)$
  $\mathcal{R}^*$: Orient all adjacent edges of F-NODE as outgoing.
  Apply the four Meek rules from [31] ($\mathcal{R}_1 - \mathcal{R}_4$) until none applies.
return $\mathcal{G}$

Algorithm 3 RCD for finite number of samples

Input: Normal dataset $\mathcal{D}$, anomalous dataset $\mathcal{D}^*$, γ, α, k : Max. no. of root causes
Output: A list of root causes $\mathcal{U}$.

procedure RCD($\mathcal{D}, \mathcal{D}^*, \gamma, \alpha, k$)
  $\mathcal{U} \leftarrow$ Set of variables of $\mathcal{D}, \mathcal{D}^*$.
  while $|\mathcal{U}| > k$ & $|\mathcal{U}| > \gamma$ **do**
    $\mathcal{S} \leftarrow$ A random partitioning of $|\mathcal{U}|$ into subsets of size γ.
    $R \leftarrow \emptyset$
    for all $S \in \mathcal{S}$ **do**
      $G \leftarrow$ Ψ-PC($\mathcal{D}[S], \mathcal{D}^*[S], \alpha$) # Ψ-PC constructs a graph on $S \cup \{\text{F-NODE}\}$.
      $R \leftarrow R \cup Ne_G(\text{F-NODE})$ # Extract neighbors of F-NODE.
    $\mathcal{U} \leftarrow R$.
return TOPK($\mathcal{D}[\mathcal{U}], \mathcal{D}^*[\mathcal{U}], k$) # Get top-$k$ neighbors of F-NODE.

In most of the RCA literature, the output of the algorithm is an ordered list of potential root causes of size k, rather than a singleton. Here $k << n$ and n is the total number of microservices. We create an ordered list of potential root causes by running Ψ-PC multiple times with different α.

The Ψ-PC algorithm takes a hyperparameter α as an input which it uses this to compare the p-values of CI tests to decide if two variables are independent. If the p-value is greater than α then we mark those two variables as independent. In this way, a strict (small) value of α will result in a sparse graph whereas a relaxed (large) value of α will produce a dense causal graph. We create an ordered list of k potential root causes by running Ψ-PC multiple times with different values of α and ordering the neighbors of F-NODE.

To create an ordered list of root causes, we modified the second phase of the hierarchical learning algorithm. After running Ψ-PC on all the subsets, F-NODE may have more than one neighbor. In this case, we need to decide which one is more likely to be the root cause. We achieve this by choosing the neighbor that is least likely to be the root cause, i.e., a node that would not have been the neighbor of F-NODE only if we had chosen a strict value for alpha. Once we identify that node, we put it at the end of the list, remove it from the neighbors of F-NODE and repeat the process until F-NODE is isolated.

On the flip side, there might be some cases where $k < |Ne_G(\text{F-NODE})|$. In those cases, we will need to find more potential root causes to build a list of k root causes. For this, we start RCD with a really strict value of alpha (mostly 0.001). If we get at least k neighbors of F-NODE, we stop the execution, order the neighbors based on their p-values, and choose the top k nodes from the list to be the top k root cause candidates. However, if the neighbors of F-NODE are less than k then we rerun the hierarchical learning algorithm with a strict value of alpha and repeat the whole process with a new value of alpha until we get k potential root causes. The top-k algorithm is shown in Algorithm 4.

Algorithm 4 Top-k Root Causes

Input: Normal dataset D, anomalous dataset D', Ψ-PC (.) [19], k : Max. no. of root causes
Output: An ordered list of top-k neighbors of F-NODE.

$\alpha_0 \leftarrow 0.001, \delta \leftarrow 0.1, \tau \leftarrow 1$
procedure TOPK(D, D', k)
  $\mathcal{L} \leftarrow \emptyset, \alpha \leftarrow \alpha_0$
  while $\alpha < \tau$ & $|\mathcal{L}| < k$ **do**
    $G \leftarrow \Psi\text{-PC}(D, D', \alpha)$
    $\mathcal{N} \leftarrow \{x | x \in Ne_G(\text{F-NODE}), x \notin \mathcal{L}\}$ # List of new neighbors of F-NODE.
    for all $N \in \mathcal{N}$ **do**
      $\mathcal{L} \leftarrow \mathcal{L} \cup N[argmax(max(Pv_G(N)))]$ # Sort the neighbors
    $\alpha \leftarrow \alpha + \delta$
return $\mathcal{L}[0 : k]$

Implementation and Test-bed Setup

We implemented Algorithm 3 and Ψ-PC in Python using the causal-learn package. For the localized learning algorithm, we re-implemented the first phase of Ψ-PC to generate the skeleton graph. Furthermore, we also added the feature to store the p-values of CI tests that we use to rank the root causes.

We use the Chi-squared test to check the independence between the two variables. To generate synthetic data, we use pyAgrum with a randomly generated DAG by controlling the number of nodes and the maximum in-degree. Next, we randomly populate the conditional probability tables for all the nodes. Using these conditional probability tables, we draw a specific number of samples for the normal dataset. To simulate an intervention on a node, we first select a node at random and randomly modify its conditional probability table thus modifying its probability distribution.

For Sock-shop, we created a cluster of microservices on Amazon Web Services (AWS) using Kubernetes[7]. The cluster consists of eight t2.xlarge machines (4 vCPU, 16 GB memory) running Ubuntu 18.04. One of these machines is the master node, whereas the remaining are worker nodes.

[7] https://kubernetes.io/

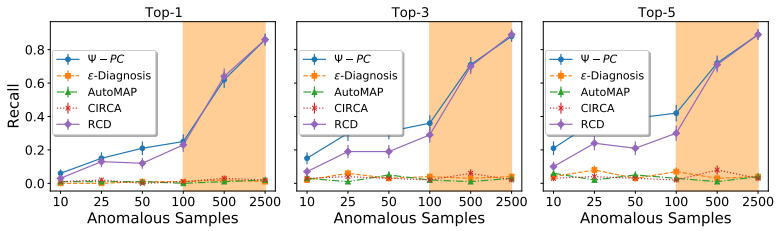


Figure 4: Sample Efficiency of different root cause localization algorithms. As the number of samples grow, the top-k recall of RCD and Ψ-PC can starts to increase. That is to say, RCD can provide the same benefits as Ψ-PC while reducing the execution time significantly.

The master node deploys the Sock-shop application on the worker nodes and routes the incoming traffic to the correct microservice. Other than these virtual machines, we have one more t2.xlarge instance running locust to send the traffic to the Sock-shop application.

On top of the Sock-shop application, we have another important tool, Prometheus. We configured it to collect the metric data from the microservices once every second. We collect the workload, CPU usage, memory usage, error counts, and latency for every microservice. In total, we collect 38 metrics from Sock-shop application (not all the microservices expose all metrics such as error counts). Finally, we use this data to generate a time series dataset of the system during the normal and anomalous states.

We injected two types of failures, CPU hog and memory leak, in different microservices of Sock-shop by running `stress-ng`. We accomplish this by modifying the Docker images of all the microservices to install `stress-ng` and then running it on the container of the targeted microservice. With this setup, we injected failures in all of the major microservices of Sock-shop (carts, catalogue, orders, payment, and user).

Sock-shop exposes a lot of metrics but not all of them are equally useful. Some of the metrics, such as the memory utilization of an inactive microservice, might remain constant during the experiment. A general solution for this problem is to remove constant variables from the data in the pre-processing step. We accomplish this by removing a metric that has a standard deviation below 1. Moreover, all of the metrics produce continuous values but the Chi-squared test only works on categorical data. Hence, we discretize the variables into a specific number of *bins* by using K-means clustering. For all our experiments with Sock-shop, we used 5 number of bins.

Sample Efficiency

Timely fix of failures in a large scale cloud-based application is of utmost importance. In other words, the root cause of the failure needs to be detected as early as possible. Consequently, one would expect a large amount of data for the normal state of the system however, the number of samples for the anomalous dataset will be quite limited. To check the applicability of any algorithm for RCA, we need to see how sample efficient that approach is. To that end, we performed an experiment by varying the number of samples and checking the top-k recall of different algorithms. For this experiment, we kept the number of nodes and number of samples for normal dataset constant (100, and 10K respectively) and varied the number of anomalous samples. Figure 4 illustrates the results.

The first thing to note here is that, in most cases, RCD is as effective in detecting the root cause as Ψ-PC even with limited number of samples. More specifically, with only 500 anomalous samples, RCD can find the root cause in top 5 with the recall of 71% (as compared to 77% for Ψ-PC). Whereas AutoMAP and ϵ-Diagnosis can only detect about 4% and 1.1% of the root causes respectively.

Multiple Root Causes

Ψ-PC, similar to Ψ-FCI, is not limited to only finding a single interventional target as the proposed approach in [19] can find multiple interventional targets from a single dataset. Accordingly, we performed an experiment to analyze the ability of RCD to detect multiple root causes from an anomalous dataset. We kept the total number of nodes to 50 and randomly chose a subset of nodes to

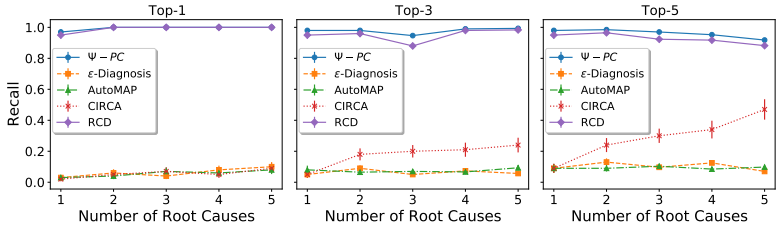


Figure 5: The top-k recall of fault localization algorithms to detect multiple root causes. RCD is as effective as Ψ-PC in detecting multiple root causes whereas other baselines fall short. CIRCA performs better than other baselines as it was able to find 47% of the root causes for some cases in the top-5.

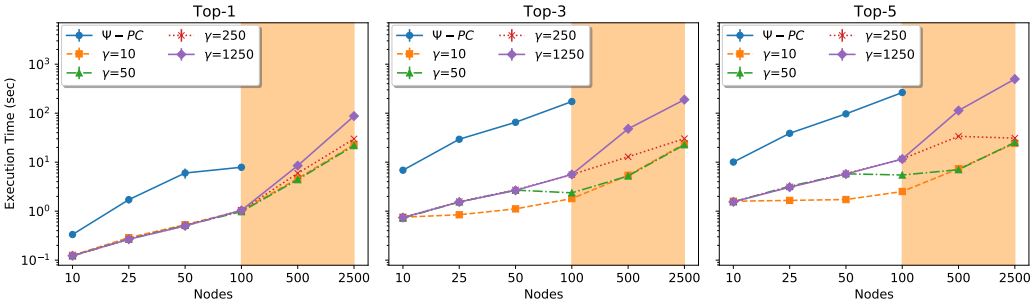


Figure 6: The execution time of Ψ-PC and RCD with different γ. It controls the size of the subsets for RCD, thus increasing γ leads to a higher execution time. However, even with higher γ and large number of nodes, RCD performs better than Ψ-PC because of its localized learning.

be the root cause. Figure Figure 5 illustrates the top-k recall of different approaches. We observe that with higher number of root causes, RCD and Ψ-PC are very likely to find a root cause in top-1. However, for top-5 recall RCD's performance start to decrease with 5 root causes (from 1.0 to 0.88). We believe this is because of insufficient number of anomalous samples (the number of anomalous samples were constant for this experiment, 10K). Even still RCD outperforms the best baseline (CIRCA) by 41%.

Sensitivity analysis of γ

RCD requires a hyperparameter, γ, which controls the size of each subset. In all of our experiments, we set γ to be 5. Here, we experiment by changing the value of γ to understand how it affects the performance of RCD. We run this experiment by controlling the number of nodes and the value of γ. The maximum in-degree of all the DAGs is 3, whereas the total number of states for every node is 6. For every experiment, we draw 10,000 samples for the normal and anomalous states of the system. We run every experiment 100 times and plot the average completion time of RCD. We also plot the execution time of Ψ-PC as the reference point. Figure 6 shows the result.

The first thing to note is that with a small number of nodes, different values of γ produce the result at about the same time. That is because when $n < \gamma$ where n is the number of nodes, the value of γ does not affect the number of subsets. In this case, there will only be one subset for the whole dataset. The effects of γ become more visible with a larger number of nodes and for the higher values of k. With the larger number of nodes, RCD creates more subsets if γ is small thus reducing the overall time. Even with higher values of γ, RCD performs better than Ψ-PC owing to its localized learning scheme. The time for higher values of k increases because we have to run multiple executions of Algorithm 4 to get a sorted list of all the neighbors of F-NODE.

Table 3: Execution time in seconds of different algorithms for detecting the root cause on data from the Sock-shop application. ϵ-Diagnosis takes less time because of its reliance on coefficient of variation but it performs poorly when the number of variables increase as shown in Figure 2a. RCD outperforms Ψ-PC and AutoMAP by 97% and 99% respectively.

	CPU hog				Memory leak			
	Ψ-PC	AMAP	ϵ-Diag.	RCD	Ψ-PC	AMAP	ϵ-Diag.	RCD
Carts	8.96	26.92	0.01	0.04	17.05	24.93	0.01	0.04
Catalogue	9.41	9.64	0.01	0.05	10.51	27.45	0.01	0.04
Orders	2.96	15.44	0.01	0.04	1.97	15.20	0.01	0.04
Payment	12.40	21.48	0.01	0.04	30.86	25.17	0.01	0.05
User	1.34	15.46	0.01	0.04	3.76	12.10	0.01	0.05

Sock-shop Experiment

We ran the Sock-shop application for 5 minutes to collect the data for the normal state of the system. After that, we injected the failure and ran the application for 5 more minutes to collect the anomalous dataset. We repeated the experiment 5 times for two different types of failures (CPU hog and memory leak) and in total, we gathered 50 datasets. To measure the performance of the root cause detection algorithm, we ran an algorithm 50 times on every dataset and quantified their top-k recall and execution time. Table 1 shows the top-k recall and Table 3 shows the execution time of different RCA algorithms.

The first thing to note here is that ϵ-Diagnosis outperforms all the other algorithms. That is because it only uses a pairwise test of coefficient of variation to find the root cause metric and microservice. Another reason it takes much less time is that for Sock-shop, we only have 38 metrics. However, as we show in our experiments, Figure 2a, the recall of ϵ-Diagnosis drops significantly because of the weak CI test for a large number of variable.

Considering RCD, we observe that it outperforms Ψ-PC and AutoMAP. More specifically, it reduces the time to find the root cause by 91% and 96% as compared to Ψ-PC and AutoMAP in the case of CPU hog failures. We get this gain by using a hierarchical approach to learn the root cause and only learning the neighborhood of F-NODE rather than learning the complete causal graph.

Related Works

Performance diagnosis in a distributed system is of paramount importance due to the higher failure chances of each component, hence affecting the system in its entirety. There has been significant research work not only in academia [41, 8, 53] but in the industry [51, 52] as well devoted to the topics of identifying and localizing root causes for the system failures. Works on root cause analysis can broadly be segregated into the following methods:

Log-based. These works make use of system log information to investigate the causes of anomalies and failure of a system [13, 55, 62]. A general approach to log analysis involves parsing the raw text data to learn historical patterns, and leveraging the patterns learned to detect anomalies. [13] uses finite state automata to represent the patterns, whereas a recent study [62] performs a learning-based methodology on the features extracted from logs to detect anomalies. The log-based approaches assume that the application produces useful logs. However, in reality the quality and quantity of different modules of the same applications can be poles apart thus, making them useful only in limited space.

Trace-based. Studies like Pinpoint [7], X-Trace [12], and Microscope [24] record the execution path information and locate the culprit services. While Pinpoint and X-Trace require instrumenting the source code and a general understanding of the code by system administrators, Microscope aims at generating a service causal graph based on the trace data. Seer [15], on the other hand, uses RPC-level tracing along with low-level hardware monitoring to identify the root cause microservice, while [23, 26] leverages unsupervised trace anomaly detection and microservice localization to detect the root cause. Furthermore, GMTA [17] uses a graph-based approach to cluster the traces. It also builds an interpretable tool for understanding microservice architecture and diagnosing problems (it does not explicitly work on identifying the root causes). One common trait of most of these tools is

that they require application instrumentation and expert knowledge therefore making them difficult to use in real-world applications.

Metrics-based. These tools aim at performing system diagnosis by leveraging the value of various metrics observed. MonitorRank [20] localizes the root cause API of failures by examining the historical time series of performance metrics along with the call graph of services. Grano [51] deployed at Ebay Inc. uses a dependency graph-based approach among physical resources to perform root cause analysis. With a call graph, we can analyze the faulty and the impacted services. However, to understand why a fault occurred, we need to understand how performance metrics for each component interact with each other. Works like [53, 54, 8, 32, 41] perform root cause analysis by building causal graphs using variations of the PC algorithm at the performance metrics level. The causal discovery algorithms output a CPDAG where some edges are undirected, and hence post-processing is necessary to convert the CPDAG to a DAG. After generating a DAG, the most common approach is to run a variety of random work or graph traversal algorithms for ranking a subset of probable root causes. These tools do not require any application instrumentation or expert knowledge. However, they suffer from high execution times because of PC that tries to learn the complete underlying causal graph.

On the contrary, our proposed algorithm models the failures as an intervention on the failing node. Thus, allowing us to leverage the state-of-the-art algorithm from causal inference to learn the interventional target. Furthermore, we propose localized hierarchical learning to overcome the high execution time of PC, therefore, making it applicable in real-world applications.