Appendix

Here, we provide the detail about the following topics;

- Preliminaries about microservice, causality, and causal graph
- Correctness proof of RCD
- RCD for finite samples
- Implementation details
- Sensitivity analysis of γ
- Sock-shop experiment
- Related Work

Preliminaries

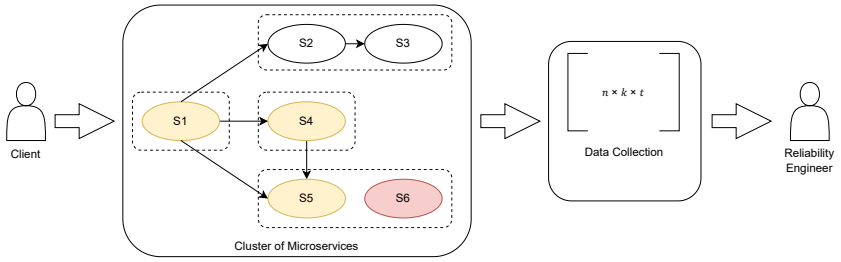

Figure 5: The workflow of a microservice-based system from the usage point of a client and SRE . Every oval is a microservice whereas every dashed rectangle is a host that hosts multiple microservices. The edges between two microservices represent the call graph. Red-colored microservices signify a failure but the effect is observed on the yellow-colored microservices. Even with the topological information about the microservices, it will be difficult for a Reliability Engineer 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 [39, 33, 15, 61]. 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 5 depicts a general web application with multiple microservices running on different hosts.

More formally, we define the microservice architecture as follows, a web application consists of a set of n microservices, $\mathcal{S} = \{s_1, s_2, ..., s_n\}$. At any given time, the monitoring tool collects m metrics from each of the microservices, i.e., $\mathcal{M}(i,t) = \{r_{i,1,t}, r_{i,2,t}, ..., r_{i,m,t}\}$ is a set of m metrics of microservice i at time instance t. To combine this all together, we have a time series dataset, $\mathcal{D} = \{\mathcal{M}(1,1), \mathcal{M}(1,2), ..., \mathcal{M}(n,t)\}$, that contains the metrics data of every microservice till time t.

Causality and Causal Graph. We define causality as a means to measure the dependence between two variables. One way to depict the causality 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 $P(v_1|v_2, S) = P(v_1|S)$.

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 [18, 30], 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 each stage. 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 stages of the algorithm. This establishes the soundness of the algorithm, that at any stage, including the final stage, root-causes will remain adjacent to the F-NODE.

RCD however is not complete.[4] 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.

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 precision 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.

Algorithm 2 Ψ-PC: Algorithm for learning the causal graph and interventional targets from observational and interventional probability distributions [18]

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 [28] ($\mathcal{R}_1 - \mathcal{R}_4$) until none applies.
return $\mathcal{G}$

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 3 RCD for finite number of samples

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

procedure RCD(D, D', γ, α, k)
  $\mathcal{U} \leftarrow$ Set of variables of D, 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($D[S], D'[S], \alpha$) # 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($D[\mathcal{U}], D'[\mathcal{U}], k$) # 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

[4] In the main paper, there is a typo calling the algorithm sound and complete.

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 (.) [18], 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 [6]. 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 [40] 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) [1] using Kubernetes[5]. 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. 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 [24] to send the traffic to the Sock-shop application.

On top of the Sock-shop application, we have another important tool, Prometheus [39]. 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` [48]. We accomplish this by modifying the Docker images of all the

[5] https://kubernetes.io/

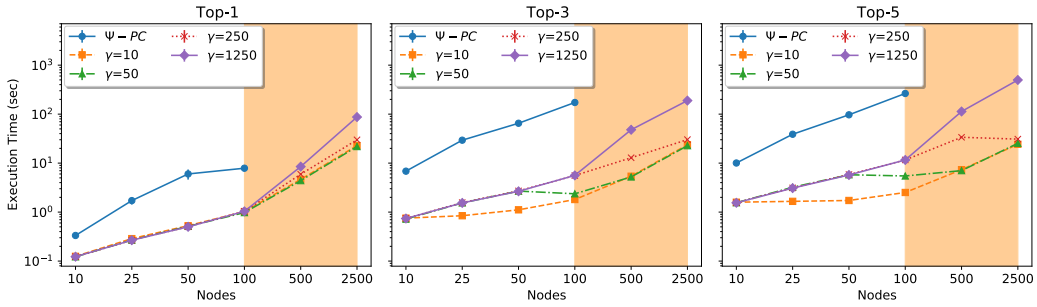

Figure 6: The execution time of Ψ-PC and RCD with different γ. It controls the size of the subsets for RCD, thus increasing γ leads to a higher execution time. However, even with higher γ and large number of nodes, RCD performs better than Ψ-PC because of its localized learning.

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.

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.

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 precision and execution time. Table 1 shows the top-k precision and Table 2 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

Table 2: 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 perfo

	CPU hog				Memory leak			
	Ψ-PC	AutoMAP	ϵ-Diagnosis	RCD	Ψ-PC	AutoMAP	ϵ-Diagnosis	RCD
Carts	8.966	17.038	0.007	0.548	17.052	14.243	0.007	0.543
Catalogue	9.410	6.139	0.006	0.393	10.512	14.572	0.005	0.326
Orders	2.964	14.361	0.004	0.739	1.972	14.579	0.007	0.659
Payment	12.403	19.832	0.005	0.670	30.857	11.807	0.006	0.490
Payment	1.338	17.259	0.005	0.490	3.763	11.645	0.006	0.697
Average	7.016	14.926	0.006	0.568	12.831	13.391	0.006	0.543

we show in our experiments, Figure 2, the precision 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 Work

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 [12, 55, 60]. A general approach to log analysis involves parsing the raw text data to learn historical patterns, and leveraging the patterns learned to detect anomalies. [12] uses finite state automata to represent the patterns, whereas a recent study [60] 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 [11], and Microscope [22] 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 [13], on the other hand, uses RPC-level tracing along with low-level hardware monitoring to identify the root cause microservice, while [21, 23] leverages unsupervised trace anomaly detection and microservice localization to detect the root cause. Furthermore, GMTA [16] 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 [19] 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, 29, 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.