Fault Localization Using Interventional Causal Learning for Cloud-Native Applications

Saurabh Jha†, Jesus Rios†, Naoki Abe†, Frank Bagehorn†, Laura Shwartz†
†IBM Research

***Abstract*—In this work, we share our experience of using recently proposed fault localization techniques based on interventional causal learning applied in the context of cloud-native applications. We identify several assumptions that prevent successful deployment of interventional causal learning fault localization in the real-world. These assumptions directly contradict the established knowledge-base of the current causality-driven work in this domain. Based on those insights, we make the following contributions: (i) development of minimal benchmark application, CausalBench, that surfaces these challenges, and (ii) adapting the interventional causal learning technique using system insights to address these challenges. Our fault localization work outperforms the recent state-of-the-art algorithms.**

I. Introduction

Ensuring high availability and reliability of a modern cloud application is challenging. These applications are composed of hundreds to thousands of microservices packaged and executed on different containers that are dynamically created and destroyed based on the incoming load [1], [2]. Each transaction may involve several microservices, and the size of their call graphs follows a heavy-tail distribution. For example, according to a recent study [1] by a major cloud provider, 10% of the call graphs consist of more than 40 microservices. Failure in any of these microservices, if not handled, will lead to transaction failures, resulting in decreased availability. Therefore, it is critically important to *identify the faulty microservice* (i.e., localize the fault) for timely mitigation.

Distributed tracing helps to localize a particular class of faults by tracking a request as it traverses the application [3], [4]. Yet, many cloud applications still lack support for tracing, and tracing itself does not encompass all fault types. For example, omission faults —faults resulting in a software process or channel failing to perform an expected action —requires costly manual inspection and visualization [5] for debugging. Given these limitations, there is substantial ongoing research to localize faults using data-driven and intervention-based causal techniques. Data-driven techniques that only rely on observational and historical data fail to identify and localize unseen faults and are susceptible to external factors such as load and environment [6]–[13]. Most causal techniques rely on domain knowledge, such as topology, to localize faults [14]–[16]. However, obtaining topology information may not be trivial for large applications. Moreover, an application's business logic may enforce other kinds of causal relations that may not be captured via topology (refer to §III). There has been limited success in identifying causal relations from observational and historical data only [17], [18]. Given these limitations, there is a growing interest in the intervention (i.e., fault injection [12], [19]–[21])-based causal learning as they not only alleviate the shortcomings of other approaches but also are interpretable [12], [14], [22]. The theory of interventional causal learning allows one to inject a minimal number of faults to learn causal relations. The identified causal relations are then used for localizing faults. Such interventional causal learning-driven fault localization is guaranteed to converge under some assumptions [23], [24]. In this paper, we highlight the practical challenges in adopting interventional causal learning for fault localization. Specifically, we identify assumptions that get violated in practice due to significant gap in mapping causal learning to the problem of fault localization and proposes a methodology to address those challenges.

This paper addresses the problem of fault localization and make the following contributions:

(a) Based on our experience with interventional causal techniques; we identify limiting assumptions which may lead to significant degradation in fault localization accuracy.
(b) We created a micro benchmark application called CausalBench that is specifically architected to surface adverse effects of those assumptions in real-world applications.
(c) We proposed a methodology that addresses the above-mentioned assumptions to improve fault localization accuracy.
(d) We evaluated our fault localization technique on CausalBench as well as Robot-shop application by forcefully injecting faults into a microservice under different load conditions. Our proposed methodology significantly outperformed other techniques [23], [24], and achieved an accuracy of 1.0 and 0.84, and informativeness (refer to §VI-A for definition) of 0.82 and 0.8, for CausalBench and Robot-shop applications, respectively.

II. Background

A. Problem Addressed — Fault localization

In this paper, we focus on the problem of fault localization [7], [8], [13], [15], [25]–[29], which in the context of a cloud-native application boils down to identifying the faulty microservice. This is an important problem because a fault in a microservice may lead to the failure of application requests directed towards this faulty microservice. Moreover a fault in one microservice can cause errors in other microservices effectively leading to stalling or complete failure of the application. Therefore, quickly localizing the failure is critical.

A popular approach is to learn fault propagation patterns to develop faults fingerprints and use them to identify the fault and its location. Fault propagation depends on the causal relations in the application, i.e., code written by the developers. Learning these relations requires both static and dynamic analysis of the code; however, observability tools in Cloud do not have access to the code and even when the code is available, doing such analysis is difficult due to large heterogeneity in use of programming languages, runtimes, and third party services (such as databases and message broker systems among others). Thus, it is important to learn as many of these causal relations as accurately as possible using automated learning techniques to localize faults efficiently and correctly in production.

In this work, we use interventions to learn the causal relations [30]–[32]. The key insight is that interventions in the

2833-292X/24/$31.00 ©2024 IEEE
DOI 10.1109/DSN-S60304.2024.00040

form of fault injections will change the behaviour of the targeted microservice, and hence, any dependent microservece will also be affected. These effects can be measured by the change in distribution of observed metrics, which under a controlled userflow allow us to identify causal relations. In contrast, non-interventional approaches rely on observed historical data in which the fault that was in the application at the moment the data was collected is typically unknown, making it very challenging to infer causal relations from this kind of data. See [30] for a more theoretical understanding.

B. Fault model.

In this paper, we consider HTTP faults. These faults can be caused by code bugs in the microservice, performance issues, or infrastructure failures (e.g., pod network failure). Furthermore, we assume that the faults are active for a sufficiently long time such that the observations collected are statistically meaningful.

III. Challenges

Fault propagation depends on the node on which the fault occurs, the type of fault, and the application logic (including its topology). These causal relations can only be learnt by using monitoring data such as logs, metrics and traces and comparing them in the presence and absence of the fault. Interventional causal learning techniques leverage fault injection to learn causal relations. However, we find that these techniques may not work in practice because of the following assumptions[1].

A. Fault Propagation Graphs Are Metric Invariant

Most works assume that the error propagation graph depends solely on application logic (expressed as code). However, we assert that the types of observed metrics is equally important in identifying the correct causal graph.

To illustrate this assumption, we define two different communication patterns as shown in Fig. 1 , the black edges show the caller-callee relation while red edges show the estimated causal influence based on the observed metric. In pattern 1, all services are stateless. For the sake of illustration, we encode the following business logic: every time node A is called, it invokes a request to B, which in turn invokes a request to C. In pattern 2, all services except D are stateless. For illustration, we encode the following application logic: every time node H is called, it increments the counter value by one stored in D; F continuously monitors the counter value stored in D and it decrements the counter value by one and calls node G. In other words, in a normal application execution, whenever H is called, G is also called indirectly via D and F.

Now consider two metrics when analyzing the error propagation (i) count of error logs, and (ii) count of API requests received by each microservice. Assuming that errors always propagate, it is not hard to see that whenever fault (e.g., service unavailable fault) is injected in node B, error logs are generated in node A and number of requests forwarded to node C decreases. Similarly, when fault is injected in node D, error logs are generated in node H and number of requests generated in G decreases (*omission fault* [33]). Thus, the causal relations used to estimate error propgation graphs also depends on the observed metrics. When considered in isolation these metrics may estimate causal graphs that are not consistent with one another.

[1]In this section, we deliberately exclude references to specific studies to focus attention on the underlying assumption rather than pinpointing issues in existing research.

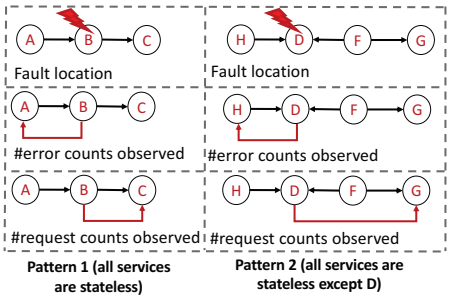

Figure 1: Causal relations depend on observed metrics & code.

B. Metric Sufficiency

To avoid the assumption of metric invariance, a learning technique may limit itself to using a single dataset modality or consider the joint distribution over all modalities. However, neither approach may be ideal. Different data modalities reveal different error propagation paths. For example, in Fig. 1, we discovered that #logs could only identify the response path (i.e., the reverse of the request path), whereas #request counts enabled us to detect omission faults. Furthermore, these metrics, as well as the algorithms used to identify causal paths, are not robust. For instance, in the aforementioned example, we assumed that errors always propagate in the direction of the response path. However, a developer might catch exceptions (or errors) and handle them without generating any error logs, or in some cases, a developer might choose not to write error logs at all; this means that the influence of a fault (or intervention) would not be observable using the count of console error logs metric, even though a causal path exists in reality. Such scenarios underscore the need for employing multiple metrics.

To avoid the pitfalls of relying on a single metric, one might consider a joint distribution over all metrics (data modalities). However, this approach may diminish the distinctiveness (i.e., identifiability) of the error propagation graph under a fault. For instance, when analyzing the joint distribution over #logs and #requests, an intervention on node B would indicate both node A and node C as causally dependent. Similarly, an intervention on node C would show both node B and node A as causally dependent, and an intervention on node A would show both node B and node C as causally dependent. Hence, failure on node B and node C as characterized by the error propagation graph (in terms of causal relations) are indistinguishable because observed data at runtime will show errors on all three nodes, i.e., A, B and, C irrespective of where the fault is injected. In the first case, node B in addition to node A and node C because node B has the fault and errors propagate to node A and node C. In the second case, node C in addition to node A ad node B because node C has the fault and errors propagate to node A and node B. Note that fault propagation graph in practice may still contain enough information to distinguish one fault from another but the causal framework as outlined here will not. Thus, one needs to be careful in making assumptions and applying a causal framework to ensure distinguishability of error propagation graphs.

rendering these faults (naturally occurring interventions) indistinguishable at runtime. Such indistinguishability can be circumvented by either conditioning on the nodes (e.g., maintaining a constant load on node C when injecting a fault on node B, as expressed through models such as do-calculus [18]) or by monitoring metrics and their corresponding causal graphs. However, conditioning on the nodes makes the intervention

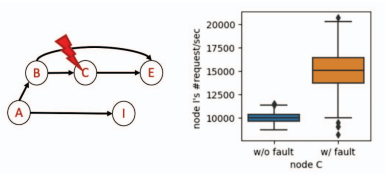

Figure 2: Confounder is intervention dependent.

mechanism more complex, and as a result, very few studies have addressed the combined issue of intervention and causality in the dependability literature [19].

C. Intervention does not change the load distribution

It is important to learn fault propagation graphs that are load invariant; i.e., nature of the load in terms of distribution of number and type of requests to avoid false positives when identifying/localizing the fault. In the literature on causality, it is well known that confounders can significantly disrupt the causal graph learning algorithm leading to spurious or incorrect edges [18]. A common example of such a confounder is the load on the application both in terms of the number of requests and the path taken by these requests. It is natural to make the simplifying assumption that the distribution of load (e.g., user requests and their paths) do not change significantly over a time window. However, in our experience, we have found that this assumption generally does not hold true in practice. Occurrence of fault (forced or natural) may change the load distribution, i.e., the intervention changes the distribution of the confounders.

For example, consider the following application topology graph shown in Fig. 2. Here, we encode the following business logic: user sends a request to node A. User can send two different types of request, one which invokes node B and another that invokes node I. Node B invokes node C or node E depending on the type of API request, and node C always invokes node E.

The steady state distribution of #requests per microservice may change depending on queuing effects. For example, if node C fails, the queue of requests on node A containing calls to C will be processed faster (because requests will return immediately). This in turn will forward more requests per second (or reduce the observed end-to-end latency of requests) to node I as seen in the boxplot in Fig. 2 despite fixing the load externally. Thus, one can conclude that node I is causally related to node C which is factually correct given the load pattern. However, if node I fails, the queue of requests on node A containing calls to I will be processed faster (because requests will return immediately). This in turn will forward more requests to node C despite fixing the load externally. *Thus, because of the load as confounder the causal relations are not consistent across fault.* Similarly, if one considers a different load pattern, one in which node A does not call node I, there will be no causal dependency between node C and node I. Hence, not modeling load (a confounder) can lead to spurious causal edges/graphs which in turn will lead to reduce in accuracy of the fault localization model.

IV. Design Overview and Methodology

We adopt the following design principles for the broader applicability.

1) The methodology must be independent of the application architecture, runtime, and technology.
2) We assume no code access.

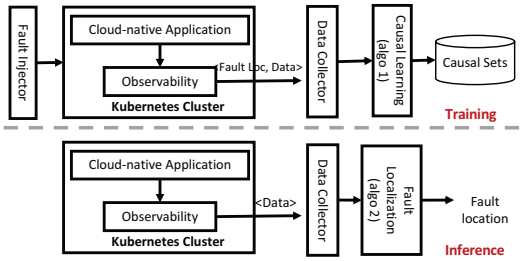

Figure 3: Design overview

3) We assume access to only monitor black-box metrics associated with a microservice, e.g., console logs and metrics such as CPU utilization.

Fig. 3 shows the overall design and implementation of our methodology. Our methodology has four important components: (i) fault injector (ii) data collector (iii) interventional causal learning, and (iv) fault localization.

A. Fault Injector and Data Collector

For fault injection and data collection we use a fault injection platform, that has been developed and implemented by our team [34]. Its design combines fault injection and data collection on the same platform, and supports the automation of use cases such as the one described in this paper. The fault injection platform supports a variety of fault categories and types. For the given use case we inject a "service unavailable" fault into the microservices. This can easily be achieved by changing the target port in the Kubernetes service configuration to point to an inactive port on the pod.

The data collection service being part of the fault injection platform eases the collection of monitoring data like logs, metrics, or traces from different monitoring applications. It hides some of the complexity of their APIs thus allowing the automation of the data collection for the presented use case.

B. Interventional Causal Learning

We formulate here our approach for solving the problem of discovering causal influences between microservices using fault injections. We have an application with a finite set of microservices S, monitored by a finite set of raw metrics $\mathcal{M}_0$. We consider here the set $\mathcal{M}$ of derived metrics defined by an SRE user as mathematical combinations of raw metrics. For example, given $M_0 =$ *"logs counts"* and $M_0' =$ *"requests received"* , a derived metric could be defined as $M = M_0 \oslash M_0'$, which would measure the *"average number of logs per request"* in a microservice and time period. The idea behind considering these derived metrics is that they should be able to deconfound causal influences. In general, the set $\mathcal{M}$ may have raw and/or derived metrics.

Each metric $M \in \mathcal{M}$ collects measurements $m(s,t)$ from each microservice $s \in S$ at successive, typically equally spaced, points in time $t \in \mathcal{T}$. We run controlled experiments in which a fault is injected in a microservice for a period of time. Let $\mathcal{T}_s$ be the set of time points $t \in \mathcal{T}$ during which an injected fault is held in microservice $s \in S$; and $\mathcal{T}_0$ the set of time points $t \in \mathcal{T}$ with no injected fault in the application. Thus, $\mathcal{T} = \cup_{s \in S} \mathcal{T}_s \cup \mathcal{T}_0$. The time series data collected for metric $M \in \mathcal{M}$ under a fault injection in $s \in S$ is $\mathcal{D}_s(M, s') = \{m(s', t), \forall t \in \mathcal{T}_s\}$, for every microservice $s' \in S$ where metric M makes sense. Similarly, $\mathcal{D}_0(M, s') = \{m(s', t), \forall t \in \mathcal{T}_0\}$ represents the time series data with no injected fault.

We want to understand how an issue in some microservice s causes anomalies in other microservices $s' \in S$, measured by same metric $M \in \mathcal{M}$. For us, an anomaly in microservice s' occurs if its distribution of values for metric M shifts, and this change cannot be explained by changes in other microservices' metrics. We use interventional causal learning to estimate $\mathcal{C}(s, M)$ the set of microservices $s' \in S$ that are causally influenced under metric M by an intervention, in the form of a fault injection, in microservice s. We know that if we observe that for every metric $M \in \mathcal{M}$ the time series datasets $\mathcal{D}_s(M, s')$ and $\mathcal{D}_0(M, s')$ are equally distributed, meaning that no anomaly is detected by any metric in s', then microservice s' cannot be causally influenced by an issue in s (at least under our observability viewpoint). However, in general, we cannot conclude that an observed distribution shift in any of these pairs of time series datasets will imply causal influence due to the possible existence of confounders, as seen in §III-C.

One option to deal with confounders is to condition on every other microservice $s'' \in S \setminus \{s, s'\}$ to block their path of causal influence. This approach, however, is computationally expensive and may still miss latent confounders that are not measured by our observability tools (for example, autoscaling actions or other SRE actions not captured by our observability tool). Also, as seen in §III-A, the use of different metrics may lead to different conclusions in terms of causal influences and confounding effects when conditioning. We take an alternative approach: we just collect the raw information about what microservices $s' \in S$ get impacted with each fault injection in s under each different metric $M \in \mathcal{M}$, and generate the sets $\mathcal{C}(s, M)$ for all $s \in S, M \in \mathcal{M}$. Algorithm 1 computes these sets by judging that $s' \in \mathcal{C}(s, M)$ when the datasets $\mathcal{D}_s(M, s')$ and $\mathcal{D}_0(M, s')$ are statistically not equally distributed. That is, if the distribution of metric M values shifts when a fault is injected in microservice s. We use the Kolmogorov–Smirnov (KS) test to decide if the hypothesis that two observed time series datasets are equally distributed is true or false. As we will see empirically in §VI-A, these sets are what we need for our fault localization heuristic proposed in §IV-C.

Algorithm 1 Fault injection-driven causal learning

Input: Set of microservices S
Input: Set of user-defined monitoring metrics $\mathcal{M}$
Input: Lengths of $\mathcal{T}_0$ and $\mathcal{T}_s$, for all $s \in S$
Collect $\mathcal{D}_0(M, s')$ for all $M \in \mathcal{M}$ and $s' \in S$
for all microservice $s \in S$ **do**
  Inject fault into s during time period $\mathcal{T}_s$
  Collect $\mathcal{D}_s(M, s')$ for all $M \in \mathcal{M}$ and $s' \in S$
  for all $M \in \mathcal{M}$ **do**
    $\mathcal{C}(s, M) = \{s\}$
    for all $s' \in S - s$ **do**
      $\widehat{F}_s(x) = \textbf{freq}(\{m \leq x : m \in \mathcal{D}_s(M, s')\})$
      $\widehat{F}_0(x) = \textbf{freq}(\{m \leq x : m \in \mathcal{D}_0(M, s')\})$
      if $F_s \neq F_0$ **then**
        $\mathcal{C}(s, M) = \mathcal{C}(s, M) \cup \{s'\}$
Return: $\mathcal{C}(s, M)$, for all $s \in S, M \in \mathcal{M}$

C. Fault Localization

We propose the following majority voting heuristic implemented by Algorithm 2 to localize faulty microservices. Periodically, we collect during production time series data $\mathcal{D}(M, s)$ for each metric $M \in \mathcal{M}$ and microservice $s \in S$. Then, we compute for each metric in $M \in \mathcal{M}$ the set of microservices $A(M)$ that behave anomalously with respect to such a metric. An anomaly in a microservice $s \in S$ is detected by comparing the empirical distributions of $\mathcal{D}(M, s)$ and $\mathcal{D}_0(M, s)$. We then emit a vote for the microservice s^* whose set $\mathcal{C}(s^*, M)$ of anomalous microservices seen when a fault was injected in microservice s is the closest to the set $A(M)$ of anomalous microservices observed in production with respect to metric M. Thus, each metric M produces a vote for a microservice, the one that seems to better explain the observed micrpservice anomalies under such a metric. Our heuristic then predicts the microservice with the most votes as the most likely cause of the observed anomalies in production for the period under consideration.

Algorithm 2 Fault localization

Input: Set of microservices S
Input: Set of user-defined monitoring metrics $\mathcal{M}$
Input: $\mathcal{D}_0(M, s)$ for all $M \in \mathcal{M}$ and $s \in S$
Input: $\mathcal{C}(s, M)$, for all $s \in S, M \in \mathcal{M}$ (Algorithm 1)
Input: $\mathcal{D}(M, s)$ for all $M \in \mathcal{M}$ and $s \in S$
Initial: $\mathcal{V}(s) = 0 \; \forall s \in S$
for all $M \in \mathcal{M}$ **do**
  $A(M) = \emptyset$
  for all $s \in S$ **do**
    $\widehat{F}_0(x) = \textbf{freq}(\{m \leq x : m \in \mathcal{D}_0(M, s)\})$
    $\widehat{F}(x) = \textbf{freq}(\{m \leq x : m \in \mathcal{D}(M, s)\})$
    if $F \neq F_0$ **then**
      $A(M) = A(M) \cup \{s\}$
  $s^* = \arg\max_s |A(M) \cap \mathcal{C}(M, s)|$
  $\mathcal{V}(s^*) = \mathcal{V}(s^*) + 1$
Return: $\arg\max_s \mathcal{V}(s)$

V. Experiment Setup

In this section, we describe the testbed and the benchmark applications used to evaluate our proposed intervention-based causal learning methodology.

A. Testbed

We created our experimental environment on a Kubernetes cluster running in a cloud environment. We deployed two microservice applications, CausalBench and Robot-Shop, consisting of nine and twelve microservices, respectively. We created a load-generation service using Locust [35] to generate various application userflows to ensure that these userflows cover all microservices. Our load-generation service, using one replica by default, emulates ten users to maintain a request throughput of fifty for each application userflow. The load-generation service can be scaled arbitrarily to increase the load by increasing the number of replicas associated with this service.

We integrated our solution into the fault injection platform described in §IV-A, that allows us to inject faults into an application's microservices without having to make any source code changes. In our experiments, we inject one fault at a time in each microservice covered by our userflows for each application, and ran the userflows for ten minutes (a configurable parameter) to get statistically meaningful data for statistical test. After running the userflows with exactly one fault injected in one of the microservices, we remove the fault from the application before injecting a fault in another microservice.

We inject *"http-service-unavailable"* faults using our fault injection platform capabilities. However, our methodology is not dependent on a specific fault type, just that faults propagate.

We monitor these microservices and collect metric data on CPU (container_cpu_user_seconds_total) and network (container_network_receive_packets_total, container_network_transmit_packets_total) using cAdvisor [36] and Prometheus [37]. We use Python Kubernetes API to directly collect message logs from each container (which is equivalent to executing `kubectl logs` on each container). Unlike previous work [23] which filters these messages to extract only error logs, we keep all messages since filtering error messages requires significant domain knowledge of the application itself. These messages are aggregated every thirty seconds to generate 'msg rate' metric. All metrics are smoothed by aggregating them using a hopping window to create overlapping sixty second windows which are created every thirty seconds. These telemetry datasets, including logs, are collected for each microservice with and without any faults for ten minutes to get the normal observational dataset and interventional dataset, respectively.

Why not use all metrics collected by cAdvisor? cAdvisor collects hundreds of metrics for each container executing the microservice. However, as discussed in §III-C, these metrics are confounded and correlated. For example, the request rate (or received bytes) measured for a container is directly related to (i) requests sent by other microservices, and (ii) the processing time for each request (which can be indirectly approximated in terms of CPU utilization) by that container. In this example, request rate and received bytes are correlated, whereas received bytes (or request rate) is confounded with the CPU utilization. Thus, arbitrarily using all metrics can skew (i) the majority voting procedure (used by algorithm 2) because of correlation and (ii) size of the causal sets because of confounding; thereby, making it important to carefully choose the metrics used for causal inference-based fault localization techniques.

To address these problems, we create a set of derived metrics from the raw metrics to eliminate correlation and confounding. In essence, we categorize our metrics into two categories (i) dependent metrics and (ii) independent metrics. Independent metrics are those that are independent of the program and controlled by external factors such as sent requests (a measure that can be collected via service mesh such as Istio [38] or approximated using network-level metrics such as received bytes). Dependent metrics are those that are influenced by the independent metrics (e.g., CPU utilization depends on number of requests that the microservice has to process). The derived metrics are created by dividing a dependent metric by an independent metric for all dependent and independent metrics.

B. Benchmark Applications

Our methodology and insights draw upon extensive knowledge from real-world applications. While it would be beneficial to disclose the specifics of these applications, such information is proprietary and confidential. Therefore, we have distilled this knowledge into a microbenchmark named CausalBench. We demonstrate the effectiveness of our approach using CausalBench, as well as other open-source benchmarks, to validate our findings and contribute to the broader research community.

CausalBench. We developed a CausalBench benchmark created specifically to reveal the challenges identified in §III.

Fig. 4 shows the architecture of our application. Here each node on the graph is a microservice, and each directed edge

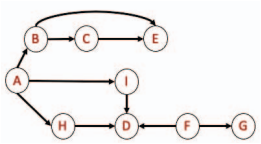

Figure 4: The topology graph of the CausalBench application

shows the caller-callee relation. All nodes are: (i) stateless except for node D which is a stateful redis service; (ii) implemented in Python; (iii) flask-based web services that expose ports and APIs for services to communicate, except for node F, and (iv) execute small compute tasks by generating a random string and calculating its base64 encoding. Our causal bench is written to support the following request flows:

(a) User calls API *path_bce* through a web request (i.e., http://A/path_bce). The *path_bce* API automatically generates an API request, *path_ce*, to node B; which in turn automatically generates an API request, *path_e*, on node C; and which in turn automatically generates an API request, /, on node E. When logging is enabled, node E writes an info log message (`I am okay!`) for every hundredth requests.

(b) User calls API *path_be* through a web request (i.e., http://A/path_be). The *path_be* API automatically generates API request, *path_e*, on node B; which in turn automatically generates an API request, /, on node E.

(c) User calls API *path_hd* through a web request (i.e., http://A/path_hd). The *path_hd* API automatically generates API request, /, on node H. In response, to this request, node H calls D (a redis service) to increment the value of counter called *items* by one.

(d) User calls API *path_id* through a web request (i.e., http://A/path_id). The *path_id* API automatically generates API request, /, on node I. In response, to this request, node I calls D (a redis service) to increment the value of another counter called *dummy* by one.

(e) Node F continuously (in an infinite loop) monitors the counter *items* by connecting to D. If the value of items is greater than one, F decrements the counter by one and each time it decrements the value by one it calls the API, /, on node G. F also writes a log to the console whenever it has finished processing hundred items and a log when there are no items to process for more than 30 seconds.

Robot-shop. Robot-shop [39] is an open-sourced microservices application of a simple e-commerce storefront. The whole application is built using 13 microservices, written on several programming languages and runtime frameworks such as AngularJS, NodeJS, Nginx, Java, Python, Golang, MongoDb, RabbitMQ, Redis, MySQL.

VI. Results

Here we evaluate the results of fault localization. We use algorithm 1 to train our model on the metric data collected with load of 1×. We evaluate the efficacy of the model on load of 1× and 4×. Note that we conducted separate experiments to collect train and test datasets for load $= 1\times$. Most recent work [23], [24] do not evaluate the model efficacy across different load distributions.

A. Fault Localization Efficacy

In order to assess the efficacy of our algorithm in finding the correct fault locations, we measure for each condition both the **accuracy** (the percentage of injected faults that are

correctly localized by our algorithm's output, i.e., an estimated set of candidate root causes) and **informativeness** (the ratio of microservices excluded by the estimated fault location set, measured as $(n-x)/(n-1)$, where n is the total number of microservices under consideration, and x is the size of the estimated fault location set). Thus, the more exclusions in the set, the more informative the estimated set is: a value of 1.0 indicates the prediction consists of only one location, and a value of 0 indicates the estimated set is as large as the total number of candidate locations.

Table I shows these results for various load configurations for CausalBench and Robot-shop. Our results show that the methodology produces accurate results, achieving an accuracy of one, when the distribution of the load is similar for training and test dataset (obtained during deployment). However, the accuracy does degrade when scaling the load by 4×. This degradation hints that our heuristics for removing correlation and confounding were not perfect.

Table I: Fault localization accuracy and informativeness

	Load	Accuracy	Informativenss
CausalBench	1×	1.00	0.82
	4×	0.84	0.80
Robot-shop	1×	1.00	0.80
	4×	0.81	0.88

B. Metrics Role in Addressing Fault Localization Challenges

We evaluate the effectiveness of intervention-based causal learning algorithms as is without addressing the challenges identified in §III. Table II shows how the informativeness decreases when we only use (i) one metric type such as error log rate (as was done in [23])[2] and (ii) raw metrics or derived metrics. Recall from §V, derived metrics are heuristic-driven metrics used for removing the effects of confounding and correlation among metrics themselves. The results indeed show the following real-world aspects and challenges of using causal learning using metrics: (i) one metric alone is not sufficient for fault localization and (ii) correlation and confounding among metrics can significantly degrade the efficacy of the causal learning techniques.

Table II: Informativeness (i) when using one or all metrics and (ii) when accounting for the confounders by use of derived metrics. Test dataset was obtained with load = $4\times$ and model training was done using load = 1×.

	Raw Metrics			Derived Metrics		
	msg rate	*cpu*	*all*	*msg rate*	*cpu*	*all*
CausalBench	0.54	0.60	0.73	0.62	0.70	0.80
Robot-shop	0.58	0.51	0.66	0.60	0.64	0.88

Finally, we show an example of how the causal sets (and the causal world) are different for different metrics. Thus, a methodology that tries to learn a single causal world (or graph or sets) using intervention learning will not succeed in mining the true causal graph. For example, Ψ-FCI algorithm [40], which is used for learning the underlying causal graph using interventional data, assumes that there is a single causal graph that enforces the relation among the nodes in the graph which is not the case as shown in the following example.

[2]In [23], the dataset consists of only error log rate whereas in our case we consider message rate which consists of both error and info log messages.

(a) Causal set extracted using msg rate when intervened on node B of CausalBench includes B, A, E. E is included because injecting fault on B stops all requests to node E which in turns prevents node E to write the `I am okay!` info message (an omission fault).
(b) Causal set extracted using CPU utilization when intervened on node B of CausalBench includes B, C, E

Note that the previously proposed causal intervention-based approach [23] only uses error rate, and hence, achieves much lower informativeness score as shown in Table II.

VII. Related Work

Fault propagation is a well studied topic in the dependability community. Previous work has mostly focused on learning fault propagation patterns using observational (and historical datasets) [6]–[9] or via fault injection-driven techniques [12], [19]–[21]. Observational models are limited to localizing faults that have already occurred as they rely on historical data to recognize fault patterns and, therefore, fail to identify novel faults. Fault injection-based techniques [12], [19]–[21] do overcome some of these limitations. However, existing techniques have mostly focused on directly learning the relation between the monitored data (e.g., logs, metrics and traces) to faults with limited understanding of causal relations and fault propagation. Techniques that do focus on extracting the causal patterns [12] and using those for fault localization do not consider confounding effects, and thus, fail to learn the true causal relation, resulting in significant degradation of accuracy.

Recent work [14], [23], [24] combines the theory of intervention-based causal learning with fault injection to limit the adverse effects of internal and external confounding in learning the true causal graphs. In [23] *only* an error rate metric is used to identify the causal graph and to localize faults. The use of error logs is problematic because errors (and exceptions) can be handled and their logging depends on software practices and on the software developer. Moreover, there is the assumption that errors only propagate in the backward direction of the call-graph, which may not always be true as is the case with omission faults. Also, correlation is used to identify causal edges that assumes that error logs are linearly related which again may not be true in practice. In [24], authors use multiple metrics to identify causal dependence overcoming the shortcomings of the [23]. However, that resulted in the challenges discussed in §III. Finally, [14] relies on an expert to define the causal structure used for fault localization. However, a main issue is that such causal structure is generally not known in advance. Application topology obtained from service meshes can substitute (or help gain) for this information but as we discuss in §III the causal structure may not always align with the structure of the application topology graph.

VIII. Conclusion

We have conveyed insights gained from our experience doing fault localization in real-world applications and encapsulate this knowledge within a microbenchmark named CausalBench. This tool is designed to aid the research community in refining and enhancing fault localization methods.

While we present a fault localization methodology that outperforms existing techniques in terms of fault localization accuracy and informativeness. Our work is just an attempt to demonstrate that straightforward techniques can significantly enhance fault localization accuracy. It is clear that further research is necessary to continue improving the performance of these algorithms.

REFERENCES

[1] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu, "Characterizing microservice dependency and performance: Alibaba trace analysis," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 412–426.

[2] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice architecture: aligning principles, practices, and culture*. " O'Reilly Media, Inc.", 2016.

[3] M. Attariyan, M. Chow, and J. Flinn, "X-ray: Automating {Root-Cause} diagnosis of performance anomalies in production software," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 307–320.

[4] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," 2010.

[5] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 243–260, 2018.

[6] L. Mariani, C. Monni, M. Pezzé, O. Riganelli, and R. Xin, "Localizing faults in cloud systems," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018, pp. 262–273.

[7] L. Wu, J. Tordsson, E. Elmroth, and O. Kao, "MicroRCA: Root cause localization of performance issues in microservices," in *Network Operations and Management Symposium*. IEEE/IFIP, 2020, pp. 1–9.

[8] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer, "Failure diagnosis using decision trees," in *International Conference on Autonomic Computing, 2004. Proceedings.* IEEE, 2004, pp. 36–43.

[9] W.-I. Kao, R. K. Iyer, and D. Tang, "Fine: A fault injection and monitoring environment for tracing the unix system behavior under faults," *IEEE Transactions on Software Engineering*, vol. 19, no. 11, pp. 1105–1118, 1993.

[10] H. Wang, P. Nguyen, J. Li, S. Kopru, G. Zhang, S. Katariya, and S. Ben-Romdhane, "Grano: Interactive graph-based root cause analysis for cloud-native distributed data platform," *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 1942–1945, 2019.

[11] B. D. Anderson, M. Deistler, and J.-M. Dufour, "On the sensitivity of granger causality to errors-in-variables, linear transformations and subsampling," *Journal of Time Series Analysis*, vol. 40, no. 1, pp. 102–123, 2019.

[12] C. Pham, L. Wang, B. C. Tak, S. Baset, C. Tang, Z. Kalbarczyk, and R. K. Iyer, "Failure diagnosis for distributed systems using targeted fault injection," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 2, pp. 503–516, 2017.

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

[14] M. Li, Z. Li, K. Yin, X. Nie, W. Zhang, K. Sui, and D. 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*, ser. KDD '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 3230–3240. [Online]. Available: https://doi.org/10.1145/3534678.3539041

[15] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou, "Sage: Practical and scalable ML-driven performance debugging in microservices," in *26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 135–151.

[16] X. Chen, M. Zhang, Z. M. Mao, and P. Bahl, "Automating network application dependency discovery: Experiences, limitations, and new solutions." in *OSDI*, vol. 8, 2008, pp. 117–130.

[17] L. M. De Campos and N. Friedman, "A scoring function for learning bayesian networks based on mutual information and conditional independence tests." *Journal of Machine Learning Research*, vol. 7, no. 10, 2006.

[18] J. Pearl, *Causality*. Cambridge university press, 2009.

[19] S. Jha, S. Banerjee, T. Tsai, S. K. S. Hari, M. B. Sullivan, Z. T. Kalbarczyk, S. W. Keckler, and R. K. Iyer, "ML-based fault injection for autonomous vehicles: A case for Bayesian Fault Injection," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019, pp. 112–124.

[20] H. Ziade, R. A. Ayoubi, R. Velazco *et al.*, "A survey on fault injection techniques," *Int. Arab J. Inf. Technol.*, vol. 1, no. 2, pp. 171–186, 2004.

[21] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[22] H. Nakama, "Inside Azure Search: Chaos Engineering," https://azure.microsoft.com/en-us/blog/inside-azure-search-chaos-engineering/.

[23] Q. Wang, J. Rios, S. Jha, K. Shanmugam, F. Bagehorn, X. Yang, R. Filepp, N. Abe, and L. Shwartz, "Fault injection based interventional causal learning for distributed applications," in *AAAI Conference on Artificial Intelligence*, 2022.

[24] M. A. Ikram, S. Chakraborty, S. Mitra, S. Saini, S. Bagchi, and M. Kocaoglu, "Root cause analysis of failures in microservices through causal discovery," in *Advances in Neural Information Processing Systems*, 2022.

[25] J. Lin, P. Chen, and Z. Zheng, "Microscope: Pinpoint performance issues with causal graphs in micro-service environments," in *International Conference on Service-Oriented Computing*, 2018, pp. 3–20.

[26] D. Liu, C. He, X. Peng, F. Lin, C. Zhang, S. Gong, Z. Li, J. Ou, and Z. Wu, "MicroHECL: High-efficient root cause localization in large-scale microservice systems," in *43rd International Conference on Software Engineering: SEIP*. IEEE/ACM, 2021, pp. 338–347.

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

[28] M. Jiang, M. A. Munawar, T. Reidemeister, and P. A. Ward, "Dependency-aware fault diagnosis with metric-correlation models in enterprise software systems," in *International Conference on Network and Service Management*. IEEE, 2010, pp. 134–141.

[29] P. Aggarwal, A. Gupta, P. Mohapatra, S. Nagar, A. Mandal, Q. Wang, and A. Paradkar, "Localization of operational faults in cloud applications by mining causal dependencies in logs using golden signals," in *International Conference on Service-Oriented Computing*, 2020, pp. 137–149.

[30] M. Kocaoglu, K. Shanmugam, and E. Bareinboim, "Experimental design for learning causal graphs with latent variables," in *Nips*, 2017.

[31] R. Addanki, S. Kasiviswanathan, A. McGregor, and C. Musco, "Efficient intervention design for causal discovery with latents," in *International Conference on Machine Learning*. PMLR, 2020, pp. 63–73.

[32] K. Bello and J. Honorio, "Computationally and statistically efficient learning of causal bayes nets using path queries," *arXiv preprint arXiv:1706.00754*, 2017.

[33] D. Powell, "Failure mode assumptions and assumption coverage," in *Predictably Dependable Computing Systems*. Springer, 1995, pp. 123–140.

[34] F. Bagehorn, J. Rios, S. Jha, R. Filepp, L. Shwartz, N. Abe, and X. Yang, "A fault injection platform for learning aiops models," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3559503

[35] Locust, https://locust.io/.

[36] cAdvisor, https://github.com/google/cadvisor.

[37] Prometheus, https://prometheus.io/.

[38] Istio, https://istio.io.

[39] I. Instana, https://www.instana.com/blog/stans-robot-shop-sample-microservice-application/, https://github.com/instana/robot-shop.

[40] A. Jaber, M. Kocaoglu, K. Shanmugam, and E. Bareinboim, "Causal discovery from soft interventions with unknown targets: Characterization and learning," *Advances in neural information processing systems*, vol. 33, pp. 9551–9561, 2020.