Fault Injection based Interventional Causal Learning for Distributed Applications

Qing Wang[1], Jesus Rios[2], Saurabh Jha[2], Karthikeyan Shanmugam[*3], Frank Bagehorn[2], Xi Yang[2], Robert Filepp[2], Naoki Abe[2], Larisa Shwartz[2]

[1] IBM Global Chief Data Office, [2] IBM Research, [3] Google Research

Abstract

We apply the machinery of interventional causal learning with programmable interventions to the domain of applications management. Modern applications are modularized into interdependent components or services (e.g. microservices) for ease of development and management. The communication graph among such components is a function of application code and is not always known to the platform provider. In our solution we learn this unknown communication graph solely using application logs observed during the execution of the application by using fault injections in a staging environment. Specifically, we have developed an active (or interventional) causal learning algorithm that uses the observations obtained during fault injections to learn a model of error propagation in the communication among the components. The "power of intervention" additionally allows us to address the presence of confounders in unobserved user interactions. We demonstrate the effectiveness of our solution in learning the communication graph of well-known microservice application benchmarks. We also show the efficacy of the solution on a downstream task of fault localization in which the learned graph indeed helps to localize faults at runtime in a production environment (in which the location of the fault is unknown). Additionally, we briefly discuss the implementation and deployment status of a fault injection framework which incorporates the developed technology.

1 Introduction

Distributed modularized applications, such as cloud-native and hybrid cloud microservices, are becoming a dominant form of enterprise applications today. These applications, particularly those running on a deeply virtualized infrastructure, are highly complex and generate massive amounts of monitoring data, posing significant challenges in ensuring application reliability. Indeed, the Uptime Institute 2022 Outage Report[1] concludes that the number of severe outages is not decreasing and the outages are becoming more costly as "scale, complexity, and secondary failures lengthen recovery times". There is a great need, therefore, for an alternative approach for application management which is not plagued by the massive operational data in production. The solution approach we propose is a framework for learning causal knowledge via fault injections in a staging environment, which is then used for application management in production.

In modularized applications, components or services communicate with one another in ways that depend on triggered userflows. Here we consider userflows triggered by human actors as well as by systems. For example, reserving a hotel room and cancelling it are two different types of user requests that trigger distinct sequences of service calls. In any application, multiple userflows are routed through various services at any point in time. When a service exhibits a fault, any services calling it may in turn throw errors, creating a storm of exceptions from multiple services, making it very difficult to localize the problem.

In machine learning terms how errors propagate in the system translates to the learning of *causal dependencies* or the possible communication links that exist between various components. In this context, the freedom to inject faults at will provides the ability to perform *interventions*. This is particularly attractive since the power of intervention allows us to bypass much of the difficulty in learning causal relationships from observational data, such as the presence of latent confounders, a well recognized issue in causal learning and inference. Since injecting faults in a production environment is undesirable, we propose a framework for active interventional causal learning from fault injections in a *staging environment*. Specifically, we adopt the methods developed in (Kocaoglu, Shanmugam, and Bareinboim 2017) and apply them to the problem of learning causal dependencies (i.e., error propagation) between application components.

Knowledge of the communication causal graph is critical in many downstream tasks in IT operations, such as active probing (Tan et al. 2019), testing (Jha et al. 2019), taint analysis (Clause, Li, and Orso 2007) and fault localization (Zhou et al. 2019). Usefulness of our methodology in such downstream tasks is demonstrated by applying it to fault localization. With that goal we devise an algorithm that leverages the causal knowledge learned in the staging environment to perform fault localization (i.e. identifying the originating faulty component) in production based on observed error patterns in that environment.

The causal learning solution we developed has been integrated into our fault injection framework, which is in use

[*] This work was done while this author was with IBM Research.

[1] https://uptimeinstitute.com/webinars/webinar-critical-update-uptime-institute-2022-outage-report/

by several divisions within IBM (see Section 7). To provide a base for possible comparative analysis, we validate and demonstrate our methodology using two publicly available benchmark microservice applications, achieving a competitive level of accuracy in learning the true edges of the underlying causal graph (actually the transitive reduction which is the minimal edge subgraph that preserves the ancestral relationships) in both cases. We also show that the learned graph can help achieve a practical level of accuracy in a downstream task of fault localization. Since we deploy our methodology in the context of microservice-based cloud-native applications, our exposition in the subsequent sections will be in terms of "microservices." We note, however, that our methodology can be applied more broadly to other highly modularized multi-component systems (and applications) as long as the individual components interact with one another through APIs.

To the best of our knowledge, this is the first application of the machinery of interventional causal learning to a problem of significant importance in applications management.

2 Related Work

Learning communication graph: Modern applications (and systems) are highly distributed, modularized, and interact with one another using network calls. To manage such complex modularized applications to ensure reliability, site reliability engineers (SREs) need to know the communication graph. The current approaches to extracting the communication graph involve techniques such as (i) tracing (Sigelman et al. 2010) which involves performance-impacting application instrumentation for each component and (ii) code annotations. Unfortunately, neither of these approaches are applicable in the context of multi-component applications in which components are developed and managed by different entities; thereby requiring an automated machine learning approach for extracting the communication graph.

Causal Learning with interventions: Earlier works on using active learning through intervention studied how many randomized interventions are needed for learning a causal DAG when there are no latents (Eberhardt, Glymour, and Scheines 2006; Eberhardt and Scheines 2007; Hauser and Bühlmann 2014). Another body of work investigated the number of interventions that would be required to learn a causal DAG assuming access to the observational Markov Equivalence class in the adaptive and non-adaptive settings (Lindgren et al. 2018; Shanmugam et al. 2015; Squires et al. 2020; Ghassami et al. 2018). All these works focused on learning causal DAGs assuming there are no latents that confound multiple observed variables. Non-adaptive intervention design for confounded causal models was studied recently in (Kocaoglu, Shanmugam, and Bareinboim 2017; Addanki et al. 2020; Bello and Honorio 2017). Our algorithm in this work is inspired by this recent line of work where single node randomized interventions on confounded systems are performed and the data collected is used to infer ancestral relationships. We show that these ideas are applicable in managing distributed applications in the cloud after suitable modification.

Fault localization: Existing approaches to fault localization in microservice applications and other distributed systems can be classified into the following categories: 1) Artificial intelligence techniques, 2) Model traversing techniques, and 3) Graph-theoretic techniques. AI techniques include rules-based or model-based methods. Rule-based methods, e.g. Liu, Mok, and Yang (1999), and Lor (1993), tend to require domain knowledge, and hence the maintenance of the rules can be laborious. Model-based approaches, e.g. neural network-based methods and others (Zhou et al. 2019; Liang et al. 2016; Qi, Yao, and Uzunov 2017), generalize better and fare better with noise and inconsistencies in the data. These approaches require a large amount of training data and "extrapolation" is a known challenge (Gardner and Harle 1997, 1998; Qi, Yao, and Uzunov 2017). Model traversing based approaches require the domain topology over the components of the application. They can be effective when the entity relationships adhere to certain restrictions but there are challenges otherwise (Kätker and Paterok 1997). Although fewer, there have been some works such as Mariani et al. (2018), Gan et al. (2021) and Zhou et al. (2019) that have applied causal approaches for fault localization using traces. The present work contrasts with these works in that it targets the *limited observability* scenario, e.g. traces are not available on customer applications to the cloud computing service provider due to the reasons stated earlier in this section.

3 Problem Definition and Methodology

Modularized modern applications, such as those using microservices architecture, consist of many loosely coupled and independently deployable code components that interact with one another to execute business logic. A representation of all possible requests between microservices that the code is architected to trigger is given by the *communication graph*. A service or cloud platform provider typically *does not* have access to code internals and therefore will not know the communication graph.

In a production environment, a microservice may experience a fault leading to the failures of the requests directed towards this faulty microservice. This may cause errors in other microservices effectively leading to stalling or failure of the application. Quickly localizing the microservice at the *root* of the problem (i.e., user request failure) is critical. This is known as the *Fault Localization Problem*. When the communication graph is unknown, localizing at runtime the faulty microservice is especially difficult. Therefore knowing the communication graph without accessing the code details becomes critically important to perform fault localization (apart from being useful in many other downstream tasks).

We inject faults into microservices in a *staging environment* (before full deployment) to generate the necessary data to solve the problem of learning the communication graph. The type of fault we consider in this paper is one in which the injected microservice fails by not responding to requests directed to it. The injected fault then propagates to any other microservice dependent on the faulty microservice (the one in which the fault was injected). For injected faults to manifest, requests to the faulty microservice must occur. We generate them by simulating one or more *userflows* consisting of a

specific set of actions taken by a user or system. This will create cascading requests across one or more microservices including the one in which the faults are injected. The userflows are simulated by capturing the real traffic from the production (in case the application is already deployed) or test-cases written by developers to validate the application's business logic. We collect data to identify which dependent microservices are impacted by the faulty microservice. In the present work, we assume access to the logs emitted by every microservice. Each log line is classified as normal or erroneous. However, we can use other indicators (e.g., metric data such as request latency) to identify which dependent microservices are impacted by the faulty microservice.

We learn the graph of causal relations between errors in microservices, which is the reverse of the communication graph, as errors propagate in the opposite direction in which microservices communicate. Our overall methodology is depicted schematically in Figure 1.

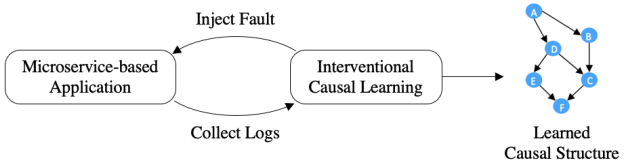

Figure 1: Our interventional causal learning methodology.

4 Algorithms and Techniques

We use a structural causal model (Pearl 2009) that explains the data generating mechanism of logs under fault injection. This is one of our principal contributions. The mapping of error log counts under fault injection in a microservice application to a structural causal model with possible confounders is crucial to apply interventional learning algorithms.

4.1 Structural Causal Model

In a structural causal model (SCM), we have a set of n *unobserved* exogenous variables $\mathcal{U} = \{u_1 \ldots, u_n\}$ and n *observed* endogenous variables $\mathcal{X} = \{x_1 \ldots, x_n\}$. Every endogenous measured variable x_i is causally related to some parental endogenous measured variables $x_j,\ j \in \mathrm{Pa}_x(i)$ and an unobserved exogenous parent variable u_i through a function $f_i(\cdot)$, i.e. $x_i = f_i(\mathbf{x}_{\mathrm{Pa}_x(i)}, u_i),\ \forall i \in [1 \ldots, n]$. The set of parents $\mathrm{Pa}_x(i)$ for every variable i defines a directed acyclic graph (DAG) called the *causal DAG*.

Mapping SCM to Microservices: Our endogenous measured variable x_i is the count of erroneous logs emitted by a microservice i in a specific time bin t. Our unobserved exogenous variables u_i represent the *userflow* variables (user inputs that are completely unobserved) that determine, at time bin t, the presence of (multiple) user requests that reach microservice i. The parents of i, $\mathrm{Pa}_x(i)$, are those microservices that i can call. Note that the edge is oriented in the opposite direction compared to the true communication graph. This is because errors propagate in the opposite direction. $f_i(\cdot)$ represents the application code detail (that is actual but not observed) that relates the count of errors thrown by microservice i, given the realization of the userflow u_i and whether there have been errors thrown by the parents in the causal graph (error propagation graph). For example, in a web based reservation application for hotels, a user might interact with the system to check availability. The user activity at time bin t would trigger requests to go between only a specific sequence of microservices. These microservices may or may not throw an error for i even if the error is thrown by $j \in \mathrm{Pa}_x(i)$, since the particular user request at i might be processed by a different parent other than j due to the specific handling of the user request by the application code (captured by the userflow variable). This is illustrated in Figure 2.

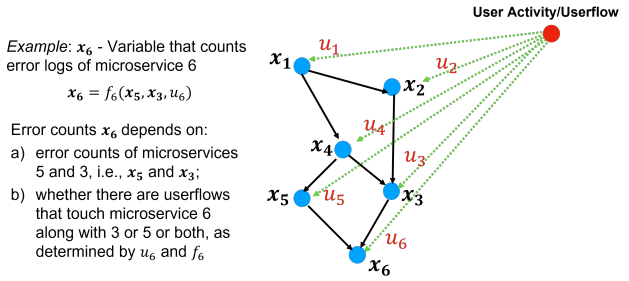

Figure 2: Structural Causal Model for microservice error log generation. The edges are reversed version of those in the true communication graph. Error counts of a microservice is a function of userflow and the error counts of its parents in the error propagation causal graph.

Confounding and Causal Insufficiency: One of the key attributes of this model is that joint distribution of userflow variables u_i that represent user requests' influence on microservice i in any time bin t are *correlated*, i.e. $P(\mathbf{u}) \neq \prod_{i \in [n]} P(u_i)$ and then the distribution does not factorize. This is because a specific user activity correlates with a subset of microservices depending on the type of user activity (e.g., booking a hotel involving payment service versus checking availability that does not involve a payment service). Therefore, in general the observations are confounded leading to a *causally insufficient system*. Relations between userflows and observed error logs are illustrated in Figure 3. Correlated error logs appear in a time bin if many user requests that take a specific communication path occur together.

Remark: Note that the graph represents the real and physical communication between microservices in an application. The userflow/user activity cannot be observed directly by a platform provider. Further, they confound error counts from logs of multiple microservices depending on the type of user activity which is an unknown distribution. The communication graph is also sparse and this sparsity comes from how the code is instrumented in the application. Usual objections to SCM modeling (Imbens 2020) such as expert evidence supporting sparsity are moot in this case as naturally the system's communication graph is sparse and it is a real logical entity that has causal implications for error propagation.

4.2 Learning the Communication Graph

We would like to learn the true edges of the error propagation causal graph (whose edges are reversed from the true communication graph). As noted before, the error log count

measurements x_i are confounded due to unobserved userflows making it a causally insufficient system[2].

We leverage interventional causal learning algorithms that target the learning of a causal DAG in the confounded case from interventions that have been developed in the causal learning literature (Kocaoglu, Shanmugam, and Bareinboim 2017; Addanki et al. 2020).

An interesting property of confounded systems is that intervening on every node in turn is *not sufficient* to recover the causal graph (Addanki et al. 2020). In fact, for single node interventions, it is necessary and sufficient to intervene on all the nodes to recover the true ancestral relationships. This is in direct contrast to the causally sufficient case where single node interventions in turn can recover the true causal graph (Hauser and Bühlmann 2014).

We estimate the minimal set of edges that would preserve the ancestral relationships in the error propagation causal graph called the *transitive reduction (TR)*. The *TR* proves to be useful in matching the pattern of errors in a production environment that helps in localization of faults.

Let the set of microservices be given by S. Let $x_s, s \in S$ be the count of error logs of a microservice in a time bin t. An epoch of fault injection lasts T_b time bins. $\mathbf{v}(s) \in \mathbb{Z}^{T_b \times 1}$ is an intervention pattern, where $\mathbf{v}(s)[t] = 1$ means that in time bin t, a fault was injected in microservice s. Over the epoch we observe the counts of error logs for a microservice s and record it in a count matrix $\mathbf{C}$, where $\mathbf{C}(t, s)$ represents the number of error logs for microservice s recorded in the log lines at time bin t.

When s' has been intervened in an epoch and s has not been intervened, a candidate causal effect of $s' \to s$ is computed by a simple correlation between s' and s and then it is thresholded at some τ:

$$corr(s', s) = \frac{\mathbf{v}(s')^T C(:, s)}{|\mathbf{v}(s')|} \tag{1}$$

Assumption 1 *The userflow distribution $P(\mathbf{u})$ and the functions $f_i's$ are such that if microservice s_j has emitted error in the time bin t then (i) if $j \in \mathrm{Pa}_x(i)$ in the error propagation graph $\mathbb{E}_{\mathbf{u}}[C[t, s_i]] \geq \tau$; (ii) if $j \notin \mathrm{Pa}_x(i)$ in the error propagation graph $C[t, s_i] = 0$ almost surely.*

In the staging environment, there is a set of user requests $\mathcal{U}$. Each userflow configuration $\mathbf{u} \in \mathcal{U}$ is run through the system in sequence one after the other with some time gap. When all user requests are run, again the same sequence is repeated until end of an epoch. In each epoch, fault is injected in a fixed microservice s and the fault is held for all time bins T_b in the epoch and logs are collected for the epoch.

Our fault injection based causal learning algorithm is described in Algorithm 1 that operates on error logs counts obtained over n epochs (where $n = |S|$ is the total number of microservices in the system). It then computes correlation 1 with all microservices s not intervened in an epoch and edges with high correlation above the threshold are retained. Then, transitive reduction of the edges is performed to obtain the minimal subgraph that preserves ancestry in the true error propagation graph. Further, all edges obtained are true causal edges (under Assumption 1) when the number of time bins is large enough. We restate the result in (Kocaoglu, Shanmugam, and Bareinboim 2017) to justify our first algorithm under Assumption 1.

[2]Note that we do not assume the knowledge of internals (or actions present) in the userflows which leads to confounding.

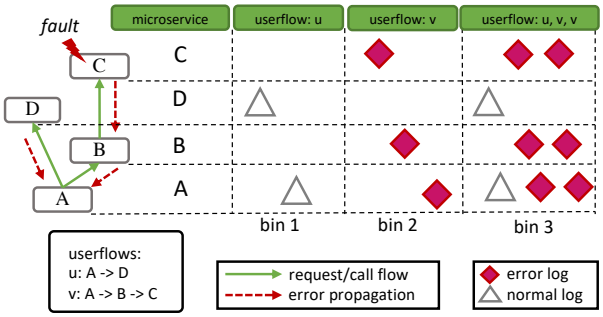

Figure 3: Illustration of userflows and error logs over time. The edges in the error propagation graph are reversed to obtain the communication graph. Microservice error counts depend on the userflows, i.e., specific user requests paths.

Theorem 1 *(Kocaoglu, Shanmugam, and Bareinboim 2017) Under Assumption 1, with threshold τ, then Algorithm 1's output is the minimal edge subgraph that preserves all ancestral relationships in the error propagation causal graph when the number of bins tends to infinity. Further, every edge in the output is present in the error propagation causal graph (reversal of which is present in the actual communication graph).*

Algorithm 1: Single-Fault Injection Causal Learning

```
Inputs: τ > 0
Initial: E = φ, G = (S, E)
for all s' ∈ S do
    Inject fault into s' based on v(s'), the all ones vector,
    and hold the injected fault for all time bins T_b.
    Filter error logs and generate C(s')
    for all s ∈ S − s' do
        if corr(s', s) > τ and (s → s') ∉ E then
            E = E + (s' → s)
        end if
    end for
end for
Return: G^tr = Transitive Reduction(G)
```

The above algorithm returns a subset of true causal edges that minimally represent all ancestral relationships in the error propagation causal graph. However, it requires as many epochs as the number of microservices in the system.

Remark: An epoch lasts for T_b time bins and a finite set of userflows $\mathcal{U}$ are run in a sequence. So, any given time bin randomly sampled induces a uniform distribution $P(\mathbf{u})$ over $\mathcal{U}$. If there is an ancestral relationship between microservices i and j in the error propagation graph and if there is an error in i, Assumption 1 can be interpreted to require that error

gets propagated to j in τ fraction of the time bins in an epoch and if there is no ancestral relationship, no error is thrown. This requirement is agnostic to variations in the system, synchronization issues with respect to the log collector during the application deployment.

4.3 Fault Localization

We now illustrate an application of our causal graph learning algorithm to one downstream task of fault localization. We recall that the fault injection stage produces a minimal edge subgraph that preserves ancestry of the original communication graph ("transitive reduction"). Our fault localization algorithm compares the observed error patterns in the logs during deployment for T_b time bins against the catalog of expected error patterns for each candidate fault location given by the transitive reduction in order to infer the likely fault location. The details of this fault localization algorithm is described in Algorithm 2. Every microservice is a candidate root cause assuming that the fault is caused by a single microservice. If microservice s was to be the root cause and if userflows touch all possible paths over T_b time bins during deployment due to enough diversity in user activity, very likely the descendants of s in the error propagation graph would all throw errors and non descendants would not. Since ancestry is preserved by the output transitive reduction G^{tr}, we can compare the pattern of errors (by comparing the normalized count of error logs of a microservice s over T_b time bins versus a threshold δ in Line 4 of Algorithm 2) with that of the indicator vector of the descendants of a microservice s. The one with the lowest structural hamming distance (SHD) is declared as the candidate root cause. It is possible that many microservices have the same minimum SHD to the thresholded error pattern observed. In that case, we output all of them as candidate root causes.

Algorithm 2: Fault Localization Using Transitive Reduction Graph for Unknown Interventional Microservice s'

1: Inputs: $\delta > 0, G^{tr}, C(s')$
2: Initial: correlation matrix $A = 0 \in \mathbb{Z}^{|S|\times|S|}$
3: **for** all $s \in S$ **do**
4: Calculate the correlation score for microservice s as $corr(s) = 1\{\mathbf{1}^T C[:, s]/T_b) > \delta\}$
5: **for** all $s, s' \in S$ **do**
6: If s' is an ancestor of s in G^{tr}, then set $A(s, s') = 1$, else $A(s, s') = 0$.
7: **end for**
8: **end for**
9: **for** all row $s \in A$ **do**
10: Compute $\mathrm{SHD}(s) = \|A[s, :] - corr[:]\|$
11: **end for**
12: Return: $\min_s \mathrm{SHD}(s)$

5 Experiment Set-Up

We set up our experimental environment on the Red Hat® OpenShift® Container Platform (OCP) in a cloud environment. OCP is an augmented Kubernetes distribution developed by Red Hat. We also created a Mezmo (former LogDNA) service instance and deployed the Mezmo agent in our OCP cluster to collect logs from every microservice and send them to a centralized repository. The Mezmo service provides an API for searching and exporting logs from this repository. We also installed in our OCP cluster the Red Hat Service Mesh, an Istio® distribution adapted to OCP, enabling us to inject faults into the microservices of an application. Finally, we deployed in our OCP cluster two publicly available microservice applications and added each one to the service mesh: **DayTrader**[3], a small application with 5 microservices that simulates an online stock trading system; and **TrainTicket**[4], a larger application with 41 microservices simulating a train ticket reservation system.

We integrated our solution into the fault injection platform (Bagehorn et al. 2022) described in Section 7, that allows us to inject faults into an application's microservices without having to make any source code changes. For injected faults to manifest, we have developed the necessary userflows for DayTrader and TrainTicket. Our userflows covered all 5 microservices in DayTrader, and 33 of the 41 microservices in TrainTicket, since we have not included administration user-role actions. In our experiments, we injected one fault at a time in each microservice covered by our userflows, for both DayTrader and TrainTicket, and ran the userflows for a few minutes. That is, the userflows are run with exactly one fault injected in one of the microservices. After that, we remove the fault from the application before injecting a fault in another microservice. In this particular work, we have injected *"http-service-unavailable"* faults using our fault injection platform capabilities. However, our methodology is not dependent on a specific fault type as long as it propagates.

At the end of our experiment, we individually collected all the logs generated by the applications, resulting in one file for each microservice's fault injection, for a total of 5 fault injection files for Daytrader and 32 for TrainTicket, since we did not inject a fault in the microservice associated with its UI. Microservices logs were classified as normal or erroneous. Fault injections propagated errors in microservice logs following their causal dependencies (see Figure 3). To prepare the log data to be consumed by our algorithms in Section 4, we partition execution time by 10ms, 100ms, and 1,000ms intervals, thus creating time bins, and count the number of error logs in each time bin, generating time series of error counts for each microservice.

6 Experimental Results

6.1 Fault Injection based Causal Learning

Here we report the results of our fault injection-based causal learning over the logs collected from the respective microservice applications, i.e., DayTrader and TrainTicket. For each microservice application, we filter the error logs with timestamps and construct the error count time series using three different time bin sizes: 10ms, 100ms, and $1,000$ms. We compute four evaluation metrics: structural hamming distance (SHD), see Tsamardinos, Brown, and Aliferis (2006),

[3] https://github.ibm.com/ocp-r2-demo
[4] https://github.com/FudanSELab/train-ticket

precision, recall, and F1-score, for each microservice application, by comparing the transitive reduction of the ground truth error propagation graph and the output of Algorithm 1 applied on the collected logs. SHD is a popular metric used to measure the distance between two graphs in terms of the difference in the edge set. We observe that both bin size and threshold τ play an important role in our causal learning as this is fundamentally an unsupervised learning problem of recovering ground truth from noisy observations. We find that, for both of the two microservice applications, the performance is better with a large bin size of $1,000$ms. Fixing the bin size as $1,000$ms, Table 1 documents the performance metrics for DayTrader and TrainTicket with different thresholds. We note that our Algorithm 1 is able to recover exactly the ground truth for the smaller application DayTrader (for a wide range of thresholds $\tau = 0.01, 0.03, 0.1$).

Application	τ	SHD	Precision	Recall	F1
DayTrader	0.01	0	1.00	1.00	1.00
	0.03	0	1.00	1.00	1.00
	0.1	0	1.00	1.00	1.00
	0.3	2	0.75	0.75	0.75
	0.4	4	0.50	0.50	0.50
TrainTicket	0.01	36	0.68	0.54	0.60
	0.03	**33**	**0.73**	**0.54**	**0.62**
	0.1	37	0.71	0.44	0.54
	0.3	52	0.40	0.08	0.13
	0.4	51	0.33	0.22	0.04

Table 1: Fault Injection Causal Learning results by comparing the transitive reduction of ground truth and the output of Algorithm 1 for **DayTrader** and **TrainTicket** with different τ values and a fixed bin size of $1,000$ ms (the best setting).

6.2 Fault Localization

We further evaluate the results of fault localization by our fault localization algorithm (refer to Algorithm 2) using the output of the interventional causal learning stage. Table 2 exhibits these results for various experimental conditions (threshold $\delta = 0.03, 0.1, 0.4$). 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 is correctly localized by our algorithm's output, i.e., an estimated set of candidate root causes) and **informativeness** (the percentage of microservices that are not in the fault location estimated set. Thus, the more exclusions in the set, the more informative the estimated set: a value of 100% 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).

7 Implementation and Deployment Path

Our causal learning solution is integrated into our fault injection framework, which is currently in use by (i) IBM product development teams to assist in the validation of their AIOps solutions, (ii) IBM Consulting to assist in customer engagements, and (ii) IBM's Sales Cloud team to support their continuous integration and deployment pipelines.

δ	Accuracy (%)	Informativeness (%)
0.03	**93.75**	**86.93**
0.1	90.63	79.55
0.4	15.63	21.69

Table 2: Fault localization performance on **TrainTicket** using the output of Algorithm 1 with different δ values and a fixed bin size of $1,000$ ms.

Our fault injection platform has been implemented as a set of FastAPI microservices written in Python and deployed to Kubernetes and Red Hat OCP clusters. The main microservices are (1) a *fault injection UI* to enable manual management, (2) a *fault injector* using different fault injection mechanisms, (3) an *arbitrator* or control plane to enable automation of fault scheduling and orchestration to create more realistic fault scenarios, and (4) a *data-collector* to collect logs, events, and traces from monitoring tools, e.g., Instana™.

The platform includes support for injecting faults in remote Kubernetes or Red Hat OCP clusters as well as virtual machines or bare metal servers running Linux. Thus it allows our solution to be broadly deployed to any class of distributed multi-component applications that use APIs to communicate with one another.

While we only use *"http-service-unavailable"* faults in this paper, our fault injection platform supports the invocation of faults from a proprietary library of faults as well as from third party libraries such as ChaosToolkit©. This way, specific faults are available to the end user through simple API calls. The richness of fault types available through existing fault injection ecosystems allows for the integration of faults targeting various cloud environments and covering the whole stack including the infrastructure, network, middleware, and application layers.

8 Conclusions

In this paper, we proposed and developed a methodology of fault injection-based interventional causal learning to help manage modern applications that are modularized into interdependent components or services (e.g. microservices), and demonstrated its execution in a cloud-native environment. Specifically, our method estimates an edge subgraph of the true communication graph that preserves ancestry, and uses that foundational knowledge for downstream management tasks. We validated its effectiveness in localizing faults in modern distributed applications for which the topology information may not be available. The developed methodology has been integrated into a fault injection framework and is in use by IBM consulting and deployed by IBM Sales Cloud team. We emphasize that as far as we are aware this is the first application of intervention design and causal learning from *interventional* data (in contrast to observational data) applied to distributed applications such as emerging cloud-native microservice-based applications.

References

Addanki, R.; Kasiviswanathan, S.; McGregor, A.; and Musco, C. 2020. Efficient intervention design for causal discovery

with latents. In *International Conference on Machine Learning*, 63–73. PMLR.

Bagehorn, F.; Rios, J.; Jha, S.; Filepp, R.; Shwartz, L.; Abe, N.; and Yang, X. 2022. A fault injection platform for learning AIOps models. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

Bello, K.; and Honorio, J. 2017. Computationally and statistically efficient learning of causal Bayes nets using path queries. *arXiv preprint arXiv:1706.00754*.

Clause, J.; Li, W.; and Orso, A. 2007. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, 196–206. New York, NY, USA: Association for Computing Machinery. ISBN 9781595937346.

Eberhardt, F.; Glymour, C.; and Scheines, R. 2006. N-1 experiments suffice to determine the causal relations among n variables. In *Innovations in machine learning*, 97–112. Springer.

Eberhardt, F.; and Scheines, R. 2007. Interventions and causal inference. *Philosophy of science*, 74(5): 981–995.

Gan, Y.; Liang, M.; Dev, S.; Lo, D.; and Delimitrou, C. 2021. 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*, 135–151.

Gardner, R. D.; and Harle, D. A. 1997. Alarm correlation and network fault resolution using the Kohonen self-organising map. In *IEEE Global Telecommunications Conference*, volume 3, 1398–1402.

Gardner, R. D.; and Harle, D. A. 1998. Pattern discovery and specification techniques for alarm correlation. In *IEEE Network Operations and Management Symposium*, volume 3, 713–722.

Ghassami, A.; Salehkaleybar, S.; Kiyavash, N.; and Bareinboim, E. 2018. Budgeted experiment design for causal structure learning. In *International Conference on Machine Learning*, 1724–1733.

Hauser, A.; and Bühlmann, P. 2014. Two optimal strategies for active learning of causal models from interventional data. *International Journal of Approximate Reasoning*, 55(4): 926–939.

Imbens, G. W. 2020. Potential outcome and directed acyclic graph approaches to causality: Relevance for empirical practice in economics. *Journal of Economic Literature*, 58(4): 1129–79.

Jha, S.; Banerjee, S.; Tsai, T.; Hari, S. K. S.; Sullivan, M. B.; Kalbarczyk, Z. T.; Keckler, S. W.; and Iyer, R. K. 2019. ML-Based Fault Injection for Autonomous Vehicles: A Case for Bayesian Fault Injection. In *49th Annual IEEE/IFIP DSN Conference*, 112–124.

Kätker, S.; and Paterok, M. 1997. Fault Isolation and Event Correlation for Integrated Fault Management. In *Proceedings of the Fifth IFIP/IEEE International Symposium on Integrated Network Management*, volume 86, 583–596.

Kocaoglu, M.; Shanmugam, K.; and Bareinboim, E. 2017. Experimental design for learning causal graphs with latent variables. In *Nips*.

Liang, C.; Benson, T.; Kanuparthy, P.; and He, Y. 2016. Finding Needles in the Haystack: Harnessing Syslogs for Data Center Management. arXiv:1605.06150.

Lindgren, E. M.; Kocaoglu, M.; Dimakis, A. G.; and Vishwanath, S. 2018. Experimental design for cost-aware learning of causal graphs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, 5284–5294.

Liu, G.; Mok, A. K.; and Yang, E. J. 1999. Composite events for network event correlation. In *Integrated Network Management VI. Distributed Management for the Networked Millennium. Proceedings of the Sixth IFIP/IEEE International Symposium on Integrated Network Management*, 247–260.

Lor, K. E. 1993. A Network Diagnostic Expert System for Acculink Multiplexers Based on a General Network Diagnostic Scheme. In *Proceedings of the IFIP TC6/WG6.6 Third International Symposium on Integrated Network Management*, volume C-12, 659–669. North-Holland.

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

Pearl, J. 2009. *Causality*. Cambridge university press.

Qi, G.; Yao, L.; and Uzunov, A. V. 2017. Fault Detection and Localization in Distributed Systems Using Recurrent Convolutional Neural Networks. In *International Conference on Advanced Data Mining and Applications*.

Shanmugam, K.; Kocaoglu, M.; Dimakis, A. G.; and Vishwanath, S. 2015. Learning causal graphs with small interventions. In *Proceedings of the 28th International Conference on Neural Information Processing Systems*, 3195–3203.

Sigelman, B. H.; Barroso, L. A.; Burrows, M.; Stephenson, P.; Plakal, M.; Beaver, D.; Jaspan, S.; and Shanbhag, C. 2010. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical report, Google, Inc.

Squires, C.; Magliacane, S.; Greenewald, K.; Katz, D.; Kocaoglu, M.; and Shanmugam, K. 2020. Active Structure Learning of Causal DAGs via Directed Clique Tree. *arXiv preprint arXiv:2011.00641*.

Tan, C.; Jin, Z.; Guo, C.; Zhang, T.; Wu, H.; Deng, K.; Bi, D.; and Xiang, D. 2019. NetBouncer: Active Device and Link Failure Localization in Data Center Networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 599–614. Boston, MA.

Tsamardinos, I.; Brown, L. E.; and Aliferis, C. F. 2006. The max-min hill-climbing Bayesian network structure learning algorithm. *Machine learning*, 65(1): 31–78.

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