Groot: An Event-graph-based Approach for Root Cause Analysis in Industrial Settings

Hanzhang Wang*, Zhengkai Wu†, Huai Jiang*, Yichao Huang*,
Jiamu Wang*, Selcuk Kopru*, Tao Xie§
*eBay, †University of Illinois at Urbana-Champaign, §Peking University
Email: {hanzwang,huajiang,yichhuang,jiamuwang,skopru}@ebay.com, zw3@illinois.edu, taoxie@pku.edu.cn

***Abstract*—For large-scale distributed systems, it is crucial to efficiently diagnose the root causes of incidents to maintain high system availability. The recent development of microservice architecture brings three major challenges (i.e., complexities of operation, system scale, and monitoring) to root cause analysis (RCA) in industrial settings. To tackle these challenges, in this paper, we present GROOT, an event-graph-based approach for RCA. GROOT constructs a real-time causality graph based on events that summarize various types of metrics, logs, and activities in the system under analysis. Moreover, to incorporate domain knowledge from site reliability engineering (SRE) engineers, GROOT can be customized with user-defined events and domain-specific rules. Currently, GROOT supports RCA among *5,000* real production services and is actively used by the SRE teams in eBay, a global e-commerce system serving more than *159 million* active buyers per year. Over 15 months, we collect a data set containing labeled root causes of 952 real production incidents for evaluation. The evaluation results show that GROOT is able to achieve 95% top-3 accuracy and 78% top-1 accuracy. To share our experience in deploying and adopting RCA in industrial settings, we conduct a survey to show that users of GROOT find it helpful and easy to use. We also share the lessons learned from deploying and adopting GROOT to solve RCA problems in production environments.**

***Index Terms*—microservices, root cause analysis, AIOps, observability**

I. INTRODUCTION

Since the emergence of microservice architecture [1], it has been quickly adopted by many large companies such as Amazon, Google, and Microsoft. Microservice architecture aims to improve the scalability, development agility, and reusability of these companies' business systems. Despite these undeniable benefits, different levels of components in such a system can go wrong due to the fast-evolving and large-scale nature of microservices architecture [1]. Even if there are minimal human-induced faults in code, the system might still be at risk due to anomalies in hardware, configurations, etc. Therefore, it is critical to detect anomalies and then efficiently analyze the root causes of the associated incidents, subsequently helping the system reliability engineering (SRE) team take further actions to bring the system back to normal.

In the process of recovering a system, it is critical to conduct accurate and efficient root cause analysis (RCA) [2], the second one of a three-step process. In the first step, anomalies are detected with alerting mechanisms [3]–[5] based on monitoring data such as logs [6]–[10], metrics/key performance indicators (KPIs) [11]–[15], or a combination thereof [16], [17]. In the second step, when the alerts are triggered, RCA is performed to analyze the root cause of these alerts and additional events, and to propose recovery actions from the associated incident [6], [18], [19]. RCA needs to consider multiple possible interpretations of potential causes for the incident, and these different interpretations could lead to different mitigation actions to be performed. In the last step, the SRE teams perform those mitigation actions and recover the system.

Based on our industrial SRE experiences, we find that RCA is difficult in industrial practice due to three complexities, particularly under microservice settings:

- ***Operational Complexity***. For large-scale systems, there are typically centered (aka infrastructure) SRE and domain (aka embedded) SRE engineers [20]. Their communication is often ineffective or limited under the microservice scenarios due to a more diversified tech stack, granular services, and shorter life cycles than traditional systems. The knowledge gap between the centered SRE team and the domain SRE team gets further enlarged and makes RCA much more challenging. Centered SRE engineers have to learn from domain SRE engineers on how the new domain changes work to update the centralized RCA tools. Thus, adaptive and customizable RCA is required instead of one-size-fits-all solutions.
- ***Scale Complexity***. There could be thousands of services simultaneously running in a large microservice system, resulting in a very high number of monitoring signals. A real incident could cause numerous alerts to be triggered across services. The inter-dependencies and incident triaging between the services are proportionally more complicated than a traditional system [15]. To detect root causes that may be distributed and many steps away from an initially observed anomalous service, the RCA approach must be scalable and very efficient to digest high volume signals.
- ***Monitoring Complexity***. A high quantity of observability data types (metrics, logs, and activities) need to be monitored, stored, and processed, such as intra-service and inter-service metrics. Different services in a system may

§Tao Xie is also affiliated with Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, China. Hanzhang Wang is the corresponding author.

produce different types of logs or metrics with different patterns. There are also various kinds of activities, such as code deployment or configuration changes. The RCA tools must be able to consume such highly diversified and unstructured data and make inferences.

To overcome the limited effectiveness of existing approaches [2], [3], [14], [16], [21]–[31] (as mentioned in Section II) in industrial settings due to the aforementioned complexities, we propose GROOT, an event-graph-based RCA approach. In particular, GROOT constructs an event causality graph, whose basic nodes are monitoring events such as performance-metric deviation events, status change events, and developer activity events. These events carry detailed information to enable accurate RCA. The events and the causalities between them are constructed using specified rules and heuristics (reflecting domain knowledge). In contrast to the existing fully learning-based approaches [3], [10], [23], GROOT provides better transparency and interpretability. Such interpretability is critical in our industrial settings because a graph-based approach can offer visualized reasoning with causality links to the root cause and details of every event instead of just listing the results. Besides, our approach can enable effective tracking of cases and targeted detailed improvements, e.g., by enhancing the rules and heuristics used to construct the graph.

GROOT has two salient advantages over existing graph-based approaches:

- ***Fine granularity*** (events as basic nodes). First, unlike existing graph-based approaches, which directly use services [25] or hosts (VMs) [30] as basic nodes, GROOT constructs the causality graph by using monitoring events as basic nodes. Graphs based on events from the services can provide more accurate results to address the monitoring complexity. Second, for the scale complexity, GROOT can dynamically create hidden events or additional dependencies based on the context, such as adding dependencies to the external service providers and their issues. Third, to construct the causality graph, GROOT takes the detailed contextual information of each event into consideration for analysis with more depth. Doing so also helps GROOT incorporate SRE insights with the context details of each event to address the operational complexity.
- ***High diversity*** (a wide range of event types supported). First, the causality graph in GROOT supports various event types such as performance metrics, status logs, and developer activities to address the monitoring complexity. This multi-scenario graph schema can directly boost the RCA coverage and precision. For example, GROOT is able to detect a specific configuration change on a service as the root cause instead of performance anomaly symptoms, thus reducing triaging efforts and time-to-recovery (TTR). Second, GROOT allows the SRE engineers to introduce different event types that are powered by different detection strategies or from different sources. For the rules that decide causality between events, we design a grammar that allows easy and fast implementations of domain-specific rules, narrowing the knowledge gap of the operational complexity. Third, GROOT provides a robust and transparent ranking algorithm that can digest diverse events, improve accuracy, and produce results interpretable by visualization.

To demonstrate the flexibility and effectiveness of GROOT, we evaluate it on eBay's production system that serves more than **159** million active users and features more than **5,000** services deployed over three data centers. We conduct experiments on a labeled and validated data set to show that GROOT achieves 95% top-3 accuracy and 78% top-1 accuracy for 952 real production incidents collected over 15 months. Furthermore, GROOT is deployed in production for real-time RCA, and is used daily by both centered and domain SRE teams, with the achievement of 73% top-1 accuracy in action. Finally, the end-to-end execution time of GROOT for each incident in our experiments is less than 5 seconds, demonstrating the high efficiency of GROOT.

We report our experiences and lessons learned when using GROOT to perform RCA in the industrial e-commerce system. We survey among the SRE users and developers of GROOT, who find GROOT easy to use and helpful during the triage stage. Meanwhile, the developers also find the GROOT design to be desirable to make changes and facilitate new requirements. We also share the lessons learned from adopting GROOT in production for SRE in terms of technology transfer and adoption.

In summary, this paper makes four main contributions:

- An event-graph-based approach named GROOT for root cause analysis tackling challenges in industrial settings.
- Implementation of GROOT in an RCA framework for allowing the SRE teams to instill domain knowledge.
- Evaluation performed in eBay's production environment with more than 5,000 services, for demonstrating GROOT's effectiveness and efficiency.
- Experiences and lessons learned when deploying and applying GROOT in production.

II. RELATED WORK

Anomaly Detection. Anomaly detection aims to detect potential issues in the system. Anomaly detection approaches using time series data can generally be categorized into three types: (1) batch-processing and historical analysis such as Surus [32]; (2) machine-learning-based, such as Donut [12]; (3) usage of adaptive concept drift, such as StepWise [33].

GROOT currently uses a combination of manually written thresholds, statistical models, and machine learning (ML) algorithms to detect anomalies. Since our approach is event-driven, as long as fairly accurate alerts are generated, GROOT is able to incorporate them.

Root Cause Analysis. Traditional RCA approaches (e.g., Adtributor [34] and HotSpot [35]) find the multi-dimensional combination of attribute values that would lead to certain quality of service (QoS) anomalies. These approaches are

effective at discrete static data. Once there are continuous data introduced by time series information, these approaches would be much less effective.

To tackle these difficulties, there are two categories of approaches based on ML and graph, respectively.

ML-based RCA. Some ML-based approaches use features such as time series information [23], [30] and features extracted using textual and temporal information [3]. Some other approaches [12] conduct deep learning by first constructing the dependency graph of the system and then representing the graph in a neural network. However, these ML-based approaches face the challenge of lacking training data. Gan et al. [10] proposed Seer to make use of historical tracking data. Although Seer also focuses on the microservice scenario, it is designed to detect QoS violations while lacking support for other kinds of errors. There is also an effort to use unsupervised learning such as GAN [12], but it is generally hard to simulate large, complicated distributed systems to give meaningful data.

Graph-based RCA. A recent survey [2] on RCA approaches categorizes more than 20 RCA algorithms by more than 10 theoretical models to represent the relationships between components in a microservice system. Nguyen et al. [21] proposed FChain, which introduces time series information into the graph, but they still use server/VM as nodes in the graph. Chen et al. [22] proposed CauseInfer, which constructs a two-layered hierarchical causality graph. It applies metrics as nodes that indicate service-level dependency. Schoenfisch et al. [24] proposed to use Markov Logic Network to express conditional dependencies in the first-order logic, but still build dependency on the service level. Lin et al. [36] proposed Microscope, which targets the microservice scenario. It builds the graph only on service-level metrics so it cannot get full use of other information and lacks customization. Brandon et al. [25] proposed to build the system graph using metrics, logs, and anomalies, and then use pattern matching against a library to identify the root cause. However, it is difficult to update the system to facilitate the changing requirements. Wu et al. [15] proposed MicroRCA, which models both services and machines in the graph and tracks the propagation among them. It would be hard to extend the graph from machines to the concept of other resources such as databases in our paper.

As mentioned in Section I, by using the event graph, GROOT mainly overcomes the limitations of existing graph-based approaches in two aspects: (1) build a more accurate and precise causality graph use the event-graph-based model; (2) allow adaptive customization of link construction rules to incorporate domain knowledge in order to facilitate the rapid requirement changes in the microservice scenario.

Our GROOT approach uses a customized page rank algorithm in the event ranking, and can also be seen as an unsupervised ML approach. Therefore, GROOT is complementary to other ML approaches as long as they can accept our event causality graph as a feature.

Settings and Scale. The challenges of operational, scale, and monitoring complexities are observed, especially being substantial in the industrial settings. Hence, we believe that the target RCA approach should be validated at the enterprise scale and against actual incidents for effectiveness.

TABLE I: The scale of experiments in existing RCA approaches' evaluations (QPS: Queries per second)

Approach	Year	Scale	Validated on Real Incidents?
FChain [21]	2013	<= 10 VMs	No
CauseInfer [22]	2014	20 services on 5 servers	No
MicroScope [36]	2018	36 services, ∼5000 QPS	No
APG [30]	2018	<=20 services on 5 VMs	No
Seer [10]	2019	<=50 services on 20 servers	Partially
MicroRCA [15]	2020	13 services, ∼600 QPS	No
RCA Graph [25]	2020	<=70 services on 8 VMs	No
Causality RCA [31]	2020	<=20 services	No

Table I lists the experimental settings and scale in existing RCA approaches' evaluations. All the listed existing approaches are evaluated in a relatively small scenario. In contrast, our experiments are performed upon a system containing 5,000 production services on hundreds of thousands of VMs. On average, the sub-dependency graph (constructed in Section IV-A) of our service-based data set is already 77.5 services, more than the total number in any of the listed evaluations. Moreover, 7 out of the 8 listed approaches are evaluated under simulative fault injection on top of existing benchmarks such as RUBiS, which cannot represent real-world incidents; Seer [10] collects only the real-world results with no validations. Our data set contains 952 actual incidents collected from real-world settings.

III. MOTIVATING EXAMPLES

In this section, we demonstrate the effectiveness of event-based graph and adaptive customization strategies with two motivating examples.

Figure 1 shows an abstracted real incident example with the dependency graph and the corresponding causality graph constructed by GROOT. The *Checkout* service of our e-commerce system suddenly gets an additional latency spike due to a code deployment on the *Service-E*. The service monitor is reporting *API Call Timeout* detected by the ML-based anomaly detection system. The simplified sub-dependency graph consisting of 6 services is shown in Figure 1a. The initial alert is triggered on the *Checkout* (entrance) service. The other nodes *Service-** are the internal services that the *Checkout* service directly or indirectly depends on. The color of the nodes in Figure 1a indicates the severity/count of anomalies (alerts) reported on each service. We can see that *Service-B* is the most severe one as there are two related alerts on it. The traditional graph-based approach [25], [30] usually takes into account only the graph between services in addition to the severity information on each service. If the traditional approach got applied on Figure 1a, either *Service-B*, *Service-D*, or *Service-E* could be a potential root cause, and *Service-B* would have the highest possibility since it has two related alerts. Such results are not useful to the SRE teams.

GROOT constructs the event-based causality graph as shown in Figure 1b. The events in each service are used as the nodes here. We can see that the *API Call Timeout* issue in *Checkout* is possibly caused by *API Call Timeout* in *Service-A*, which is

(a) Dependency graph

(b) Causality graph

Fig. 1: Motivating example of event causality graph

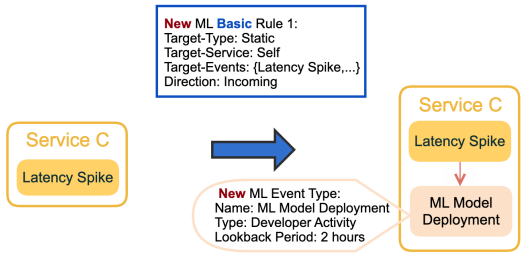


Fig. 2: Example of event type addition

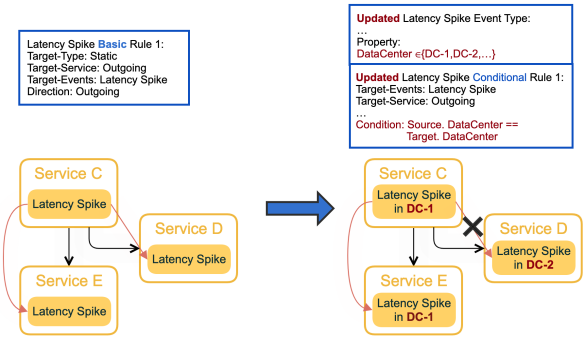


Fig. 3: Example of event and rule update

further caused by *Latency Spike* in *DataCenter-A* of *Service-C*. GROOT further tracks back to find that it is likely caused by *Latency Spike* in *Service-E*, which happens in the same data center. Finally GROOT figures out that the most probable root cause is a recent *Code Deployment* event in *Service-E*. The SRE teams then could quickly locate the root cause and roll back this code deployment, followed by further investigations.

There are no casual links between events in *Service-B* and *Service-A*, since no causal rules are matched. The *API Call Timeout* event is less likely to depend on the event type *High CPU* and *High GC*. Therefore, the inference can eliminate *Service-B* from possible root causes. This elimination shows the benefit of the event-based graph. Note that there is another event *Latency Spike* in *Service-D*, but not connected to *Latency Spike* in *Service-C* in the causality graph. The reason is that the *Latency Spike* event in *Service-C* happens in *DataCenter-A*, not *DataCenter-B*.

Figures 2 and 3 show how SRE engineers can easily change GROOT to adapt to new requirements, by updating the events and rules. In Figure 2, SRE engineers want to add a new type of deployment activity, *ML Model Deployment*. Usually, the SRE engineers first need to select the anomaly detection model or set their own alerts and provide alert/activity data sources for the stored events. In this example, the event can be directly fetched from the ML model management system. Then GROOT also requires related properties (e.g., the detection time range) to be set for the new event type. Lastly, the SRE engineers add the rules for building the causal links between the new event type and existing ones. The blue box in Figure 2 shows the rule, which denotes the edge

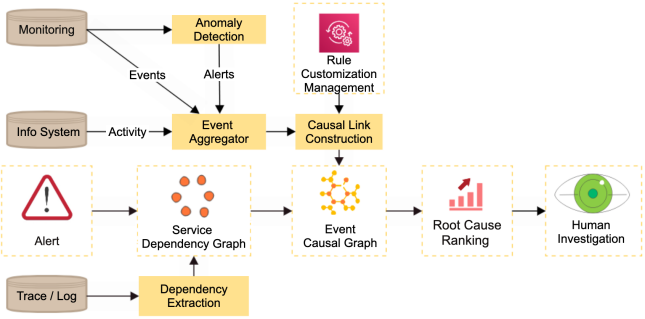


Fig. 4: Workflow of GROOT

direction, target event, and target service (self, upstream, and downstream dependency).

Figure 3 shows a real-world example of how GROOT is able to incorporate SRE insights and knowledge. More specifically, SRE engineers would like to change the rules to allow GROOT to distinguish the latency spikes from different data centers. As an example in Figure 1b, *Latency Spike* events propagate only within the same data center. During GROOT development, SRE engineers could easily add new property *DataCenter* to the *Latency Spike* event. Then they add the corresponding "conditional" rules to be differentiated with the "basic" rules in Figure 3. In conditional rules, links are constructed only when the specified conditions are satisfied.

IV. Approach

Figure 4 shows the overall workflow of GROOT. The triggers for using GROOT are usually alert(s) from automated anomaly detection, or sometimes an SRE engineer's suspicion. There are three major steps: constructing the service dependency graph, constructing the event causality graph, and root cause ranking. The outputs are the root causes ranked by the likelihood. To support fast human investigation experience, we build an interactive UI as shown in Figure 8: the service dependency, events with causal links and additional details such as raw metrics or the developer contact (of a code deployment event) are presented to the user for next steps. As an offline part of human investigation, we label/collect a data set, perform validation, and summarize the knowledge for further improvement on all incidents on a daily basis.

A. Constructing Service Dependency Graph

The construction of the service dependency graph starts with the initial alerted or suspicious service(s), denoted as I. For example, in Figure 1a, $I = \{\textit{Checkout}\}$. I can contain multiple services based on the range of the trigger alerts or suspicions. We maintain domain service lists where domain-level alerts can be triggered because there is no clear service-level indication.

At the back end, GROOT maintains a global service dependency graph G_{global} via distributed tracing and log analysis. The directed edge from nodes A to B (two services or system components) in the dependency graph indicates a service invocation or other forms of dependency. In Figure 1a, the black arrows indicate such edges. Bi-directional edges and cycles between the services can be possible and exist. In this work, the global dependency graph is updated daily.

The service dependency (sub)graph G is constructed using G_{global} and I. An extended service list L is first constructed by traversing each service in I over G_{global} for a radius range r. Each service $u \in L$ can be traversed by at least one service $v \in I$ within r steps: $L = \{u | \exists v \in I,\ dist(u,v) \leq r \text{ or } dist(v,u) \leq r\}$. Then, the service dependency subgraph G is constructed by the nodes in L and the edges between them in G_{global}. In our current implementation, r is set to 2, since this dependency graph may be dynamically extended in the next steps based on events' detail for longer issue chains or additional dependencies.

B. Constructing Event Causality Graph

In the second step, GROOT collects all supported events for each service in G and constructs the causal links between events.

1) Collecting Events: Table II presents some example event types and detection techniques for GROOT's production implementation. For detection techniques, "De Facto" indicates that the event can be directly collected via a specific API or storage. The detection either runs passively in the back end to reduce delay and improve accuracy, or runs actively for only the services within the dependency graph range to save resources.

TABLE II: List of example event types used in GROOT

Type	Event Type	Detection Technique
Performance Metrics	High GC (Overhead)	Rule-based
	High CPU Usage	Rule-based
	Latency Spike	Statistical Model
	TPS Spike	Statistical Model
	Database Anomaly	ML Model
	Business Metric Anomaly	ML Model
Status Logs	WebAPI Error	Statistical Model
	Internal Error	Statistical Model
	ServiceClient Error	Statistical Model
	Bad Host	ML Model
Developer Activities	Code Deployment	De Facto
	Configuration Change	De Facto
	Execute URL	De Facto

There are three major categories of events: performance metrics, status logs, and developer activities:

- *Performance metrics* represent an anomaly of monitored time series metrics. For example, high CPU usage indicates that the service is causing high CPU usage on a certain machine. In this category, most events are continuously and passively detected and stored.
- *Status logs* are caused by abnormal system status, such as spike of HTTP error code metrics while accessing other services' endpoints. Different types of error metrics are important and supported in GROOT, including third-party APIs. For example, Bad Host indicates abnormal patterns on some machines running the service, and can be detected by a clustering-based ML approach.
- *Developer activities* are the events generated when a certain activity of developers is triggered, such as code deployment and config change.

In Groot, there are more than a dozen event types such as *Latency Spike* as listed in the column 2 of Table II. Each event type is characterized by three aspects: $Name$ indicates the name of this event type; $LookbackPeriod$ indicates the time range to look back (from the time when the use of GROOT is triggered) for collecting events of this event type; $PropertyType$ indicates the types of the properties that an event of this event type should hold. $PropertType$ is characterized by a vector of pairs, each of which indicates the string type for a property's name and the primitive type for the property's value such as string, integer, and float. Formally, an event type is defined as a tuple: $ET =< Name, LookbackPeriod, PropertyType >$ where $PropertyType =< (string, \textit{type}_1), ..., (string, \textit{type}_n) >$ (n is the number of properties that an event of this event type holds).

Each event of a certain event type ET is characterized by four aspects: *Service* indicates the service name that the event belongs to; *Type* indicates ET's *Name*; *StartTime* indicates the time when the event happens; *Properties* indicates the properties that the event holds. Formally, an event is defined as a tuple: $e =< Service, Type, StartTime, Properties >$ where $Properties$ is an instantiation of ET's $PropertyType$.

For example, in Figure 1, the generated event for *Latency Spike in DataCenter-A* in *Service-C* would be $<$ "*Service-C*", "*Latency Spike*", *2021/08/01-12:36:04*, $<$ ("*DataCenter*", "*DC-1*"), ... $>>$.

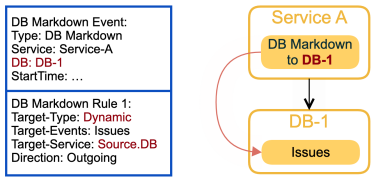


Fig. 5: Example of dynamic rule

2) Constructing Causal Link: After collecting all events on all services in G, in this step, causal links between these events are constructed for RCA ranking. The causal links (red arrows) in Figure 1b are such examples. A causal link represents that the source event can possibly be caused by the target event. SRE knowledge is engineered into rules and used to create causal links between the pairs of events.

A rule for constructing a causal link is defined as a tuple: $Rule =< Target\text{-}Type, Source\text{-}Events, Target\text{-}Events, Direction, Target\text{-}Service, Condition >$ ($Condition$ can be optionally specified). $Target\text{-}Type$ indicates the type of the rule, being either $Static$ or $Dynamic$ (explained further later). $Source\text{-}Events$ indicates the type of the causal link's source event ($Source\text{-}Events$ are listed in the names of the rules shown in Figures 2, 3 and 5). $Target\text{-}Events$ indicates the type of the causal link's target event. $Direction$ indicates the direction of the casual link between the target event and source event. $Target\text{-}Service$ indicates the service that the target event should belong to. Note that $Target\text{-}Service$ in $Static$ rules can be $Self$, which indicates that the target event would be within the same service as the source event, or $Outgoing/Incoming$, which indicates that the target event would belong to the downstream/upstream services of the service that the source event belongs to in G.

There are two categories of special rules. The first category is *dynamic* rules (i.e., rules whose $Target\text{-}Type$ is set to $Dynamic$) to support dynamic dependencies. Here $Target\text{-}Service$ does not indicate any of the three possible options listed earlier but indicates the name of the target service that GROOT would need to create. For example, live DB dependencies are not available due to different tech stacks and high volume. In Figure 5, a DB issue (DB Markdown) is shown in *Service-A*. Based on the listed *dynamic* rule, GROOT creates a new "service" *DB-1* in G, a new event "Issues" that belongs to *DB-1*, and a causal link between the two events. In practice, the SRE teams use dynamic rules to cover a lot of third-party services and database issues since the live dependencies are not easy to maintain.

The second category of special rules is *conditional* rules. *Conditional* rules are used when some prerequisite conditions should be satisfied before a certain causal link is created. In these rules, $Condition$ is specified with a boolean predicate. As shown in Figure 3, the SRE teams believe *Latency Spike* events from different services are related only when both events happen within the same data center. Based on this observation, GROOT would first evaluate the predicate in $Condition$ and build only the causal link when the predicate is true. A conditional rule overwrites the basic rule on the same source-target event pair.

When constructing causal links, GROOT first applies the *dynamic* rules so that dynamic dependencies and events are first created at once. Then for every event in the initial services (denoted as I), if the rule conditions are satisfied, one or many causal links are created from this event to other events from the same or upstream/downstream services. When a causal link is created, the step is repeated recursively for the target event (as a new origin) to create new causal links. After no new causal links are created, the construction of the event causality graph is finished.

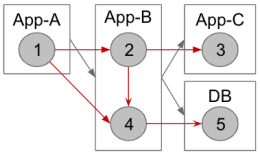
C. Root Cause Ranking

Finally, GROOT ranks and recommends the most probable root causes from the event causality graph. Similar to how search engines infer the importance of pages by page links, we customize the PageRank [37] algorithm to calculate the root cause ranking; the customized algorithm is named as GrootRank. The input is the event causality graph from the previous step. Each edge is associated with a weighted score for weighted propagation. The default value is set as 1, and is set lower for alerts with high false-positive rates.

Based on the observation that dangling nodes are more likely to be the root cause, we customize the personalization vector as $P_n = f_n$ or $P_d = 1$, where P_d is the personalization score for dangling nodes, and P_n is for the remaining nodes; and f_n is a value smaller than 1 to enhance the propagation between dangling nodes. In our work, the parameter setting is $f_n = 0.5$, $\alpha = 0.85$, $max_{iter} = 100$ (which are parameters for the PageRank algorithm). Figure 6 illustrates an example. The grey circles are the events collected from three services and one database. The grey arrows are the dependency links and the red ones are the causal links with the weight of 1. Both of the PageRank and GrootRank algorithms detect $event5$ (DB issue) as the root cause, which is expected and correct. However, the PageRank algorithm ranks $event4$ higher than $event3$. But $event3$ of *Service-C* is more likely to be the second most possible root cause (besides $event5$), because the scores on dangling nodes are propagated to all others equally in each iteration. We can see that $event3$ is correctly ranked as second using the GrootRank algorithm.

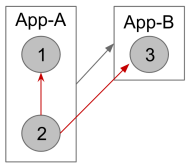
The second step of GrootRank is to break the tied results from the previous step. The tied results are due to the fact that the event graph can contain multiple disconnected sub-graphs with the same shape. We design two techniques to untie the ranking:

1) For each joint event, the access distance (sum) is calculated from the initial anomaly service(s) to the service where the event belongs to. If any "access" is not reachable, the distance is set as $d_m + 1$ where d_m is the maximum possible distance. The one with shorter access distance (sum) would be ranked higher and vice versa. Figure 7 presents an example, where *Service-A* and *Service-B* are both initial anomaly services. Since



Node	PageRank	GrootRank
1	0.11	0.10
2	0.16	0.14
3	0.18	0.22
4	0.23	0.19
5	0.31	0.33

Fig. 6: Example of personalization vector customization



Tied Score	GrootRank
1, 3 (**tied**)	3 (distance = 0 + 1)
2	1 (distance = 0 + 2)
	2

Fig. 7: Example of using access distance to untie the ranking results

GROOT suspects that $event2$ is caused by either $event3$ or $event1$ with the same weight. The scores of $event3$ and $event1$ are tied. Then, $event3$ has a score of 1 (i.e., $0+1$) and $event1$ has a score of 2 (i.e., $0+2$), since it is not reachable by *Service-B*). Therefore, $event3$ is ranked first and logical.

2) For the remaining joint results with the same access distances, GROOT continues to untie by using the historical root cause frequency of the event types under the same trigger conditions (e.g., checkout domain alerts). This frequency information is generated from the manually labeled dataset. A more frequently occurred root cause type is ranked higher.

D. Rule Customization Management

While GROOT users create or update the rules, there could be overlaps, inconsistencies, or even conflicts being introduced such as the example in Figure 3. GROOT uses two graphs to manage the rule relationships and avoid conflicts for users. One graph is to represent the link rules between events in the same service (*Same-Graph*) while the other is to represent links between different services (*Diff-Graph*). The nodes in these two graphs are the event types defined in Section IV-B. There are three statuses between each (directional) pair of event types: (1) no rule, (2) only basic rule, and (3) conditional rule (since it overwrites the basic rule). In *Same-Graph*, GROOT does not allow self-loop as it does not build links between an event and itself.

When rule change happens, existing rules are enumerated to build edges in *Same-Graph* and *Diff-Graph* based on $Target\text{-}Events$ and $Target\text{-}Service$. Based on the users' operation of (1) "remove a rule", GROOT removes the corresponding edge on the graphs; (2) "add/update a rule", GROOT checks whether there are existing edges between the given event types, and then warns the users for possible overwrites. If there are no conflicts, GROOT just adds/updates edges between the event types.

After all changes, GROOT extracts the rules from the graphs by converting each edge to a single rule. These rules are automatically implemented, and then tested against our labeled data set. The GROOT users need to review the changes with validation reports before the changes go online.

V. EVALUATION

We evaluate GROOT in two aspects: (1) *effectiveness (accuracy)*, which assesses how accurate GROOT is in detecting and ranking root causes, and (2) *efficiency*, which assesses how long it takes for GROOT to derive root causes and conduct end-to-end analysis in action. Particularly, we intend to address the following research questions:

- **RQ1.** What are the accuracy and efficiency of GROOT when applied on the collected dataset?
- **RQ2.** How does GROOT compare with baseline approaches in terms of accuracy?
- **RQ3.** What are the accuracy and efficiency of GROOT in an end-to-end scenario?

A. Evaluation Setup

To evaluate GROOT in a real-world scenario, we deploy and apply GROOT in eBay's e-commerce system that serves more than 159 million active buyers. In particular, we apply GROOT upon a microservice ecosystem that contains over 5,000 services on three data centers. These services are built on different tech stacks with different programming languages, including Java, Python, Node.js, etc. Furthermore, these services interact with each other by using different types of service protocols, including HTTP, gRPC, and Message Queue. The distributed tracing of the ecosystem generates 147B traces on average per day.

1) Data Set: The SRE teams at eBay help collect a labeled data set containing 952 incidents over 15 months (Jan 2020 - Apr 2021). Each incident data contains the input required by GROOT (e.g., dependency snapshot and events with details) and the root cause manually labeled by the SRE teams. These incidents are grouped into two categories:

- *Business domain incidents.* These incidents are detected mainly due to their business impact. For example, end users encounter failed interactions, and business or customer experience is impacted, similar to the example in Figure 1.
- *Service-based incidents.* These incidents are detected mainly due to their impact on the service level, similar to the example in Figure 5.

An internal incident may get detected early, and then likely get categorized as a service-based incident or even solved directly by owners without records. On the other hand, infrastructure-level issues or issues of external service providers (e.g., checkout and shipping services) may not get detected until business impact is caused.

There are 782 business domain incidents and 170 service-based incidents in the data set. For each incident, the root cause

TABLE III: Accuracy of RCA by GROOT and baselines

	GROOT		Naive		Non-adaptive	
	Top 3	Top 1	Top 3	Top 1	Top 3	Top 1
Service-based	92%	74%	25%	16%	84%	62%
Business domain	96%	81%	2%	1%	28%	26%
Combined	95%	78%	6%	3%	38%	33%

is manually labeled, validated, and collected by the SRE teams, who handle the site incidents everyday. For a case with multiple interacting causes, only the most actionable/influential event is labelled as the root cause for the case. These actual root causes and incident contexts serve as the ground truth in our evaluation.

2) GROOT *Setup:* The GROOT production system is deployed as three microservices and federated in three data centers with nine 8-core CPUs, 20GB RAM pods each on Kubernetes.

3) Baseline Approaches: In order to compare GROOT with other related approaches, we design and implement two baseline approaches for the evaluation:

- *Naive Approach.* This approach directly uses the constructed service dependency graph (Section IV-A). The events are assigned a score by the severeness of the associated anomaly. Then a normalized score for each service is calculated summarizing all the events related to the service. Lastly, the PageRank algorithm is used to calculate the root cause ranking.
- *Non-adaptive Approach.* This approach is not context-aware. It replaces all special rules (i.e., conditional and dynamic ones) with their basic rule versions. Its other parts are identical to GROOT.

The non-adaptive approach can be seen as a baseline for reflecting a group of graph-based approaches (e.g., CauseInfer [22] and Microscope [36]). These approaches also specify certain service-level metrics but lack the context-aware capabilities of GROOT. Because the tools for these approaches are not publicly available, we implement the non-adaptive approach to approximate these approaches.

B. Evaluation Results

1) RQ1: Table III shows the results of applying GROOT on the collected data set. We measure both top-1 and top-3 accuracy. The top-1 and top-3 accuracy is calculated as the percentage of cases where their ground-truth root cause is ranked within top 1 and top 3, respectively, in GROOT's results. GROOT achieves high accuracy on both incident categories. For example, for business domain incidents, GROOT achieves 96% top-3 accuracy.

The unsuccessful cases that GROOT ranks the root cause after top 3 are mostly caused by missing event(s). More than one-third of these unsuccessful cases have been addressed by adding necessary events and corresponding rules over time. For example, initially, we had only an event type of general error spike, which mixes different categories of errors and thus causes high false-positive rate. We then have designed different event types for each category of the error metrics (including various internal and client API errors). In many cases that GROOT ranks the root cause after top 1, the labeled root cause is just one of the multiple co-existing root causes. But for fairness, the SRE teams label only a single root cause in each case. According to the feedback from the SRE teams, GROOT still facilitates the RCA process for these cases.

TABLE IV: Comparison of GROOT results on the dataset and end-to-end scenario

	Service-based		Business Domain	
	Dataset	End-to-End	Dataset	End-to-End
Top-1 Accuracy	74%	73%	81%	73%
Top-3 Accuracy	92%	91%	96%	87%
Average Runtime Cost	1.06s	3.16s	0.98s	2.98s
Maximum Runtime Cost	1.69s	4.56s	1.14s	3.61s

Our results show that the runtime cost of applying GROOT is relatively low. For a service-based incident, the average runtime cost of GROOT is 1.06s while the maximum is 1.69s. For a business domain incident, the average runtime cost is 0.98s while the maximum is 1.14s.

2) RQ2: We additionally apply the baseline approaches on the data set. Table III also shows the evaluation results. The results show that the accuracy of GROOT is substantially higher than that of the baseline approaches. In terms of the top-1 accuracy, GROOT achieves 78% compared with 3% and 33% of the naive and non-adaptive approaches, respectively. In terms of the top-3 accuracy, GROOT achieves 95% compared with 6% and 38% of the naive and non-adaptive approaches, respectively.

The naive approach performs worst in all settings, because it blindly propagates the score at service levels. The accuracy of the non-adaptive approach is much worse for business domain incidents. The reason is that for a business domain incident, it often takes a longer propagation path since the incident is triggered by a group of services, and new dynamic dependencies may be introduced during the event collection, causing more inaccuracy for the non-adaptive approach. There can be many non-critical or irrelevant error events in an actual production scenario, aka "soft" errors. We suspect that these non-critical or irrelevant events may be ranked higher by the non-adaptive approach since they are similar to injected faults and hard to be distinguished from the actual ones. GROOT uses dynamic and conditional rules to discover the actual causal links, building fewer links related to such non-critical or irrelevant events for leading to higher accuracy.

3) RQ3: To evaluate GROOT under an end-to-end scenario, we apply GROOT upon actual incidents in action. Table IV shows the results. The accuracy has a decrease of up to 9 percentage points in the end-to-end scenario, with some failures caused by production issues such as missing data and service/storage failures. In addition, the runtime cost is increased by up to nearly 3 seconds due to the time spent on fetching data from different data sources, e.g., querying the events for a certain time period.

VI. EXPERIENCE

GROOT currently supports daily SRE work. Figure 8 shows a live GROOT's "bird's eye view" UI on an actual simple checkout incident. Service C has the root cause ($ErrorSpike$)

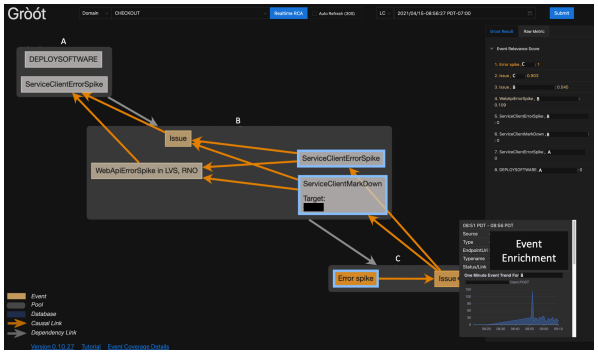


Fig. 8: GROOT UI in production

and belongs to an external provider. Although the domain service A also carries an error spike and gets impacted, GROOT correctly ignores the irrelevant deployment event, which has no critical impact. The events on C are virtually created based on the dynamic rule. Note that all causal links (yellow) in the UI indicate "is cause of", being the opposite of "is caused by" as described in Section IV-B to provide more intuitive UI for users to navigate through. GROOT visualizes the dependency and event causality graph with extra information such as an error message. The SRE teams can quickly comprehend the incident context and derived root cause to investigate GROOT further. A mouseover can trigger "event enrichment" based on the event type to present details such as raw metrics and other additional information.

We next share two major kinds of experience:

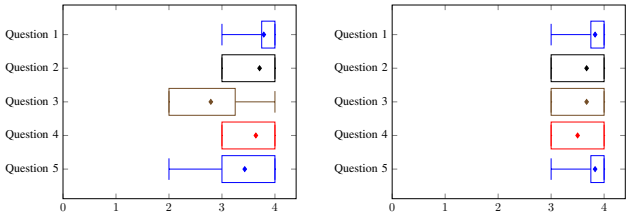
- **Feedback from GROOT users and developers**, reflecting the general experience of two groups: (1) domain SRE teams who use GROOT to find the root cause, and (2) a centered SRE team who maintains GROOT to facilitate new requirements.
- **Lessons learned**, representing the lessons learned from deploying and adopting GROOT in production for the real-world RCA process.

A. *Feedback from* GROOT *Users and Developers*

We invite the SRE members who use GROOT for RCA in their daily work to the user survey. We call them users in this section. We also invite different SRE members responsible for maintaining GROOT to the developer survey. We call them developers in this section. In total, there are 14 users and 6 developers[1] who respond to the surveys.

For the user survey, we ask 14 users the following 5 questions (Questions 4-5 have the same choices as Question 1):

- **Question 1.** When GROOT correctly locates the root cause, how does it help with your triaging experience? Answer choices: Helpful(4), Somewhat Helpful(3), Not Helpful(2), Misleading(1).
- **Question 2.** When GROOT correctly locates the root cause, how does it save/extend your or the team's triaging time? (Detection and remediation time not included) Answer choices: Lots Of Time Saved(4), Some Time Saved(3), No Time Saved(2), Waste Time Instead(1).
- **Question 3.** Based on your estimation, how much triage time GROOT would save on average when it correctly locates the root cause? (Detection and remediation time not included) Answer choices: More than 50%(4), 25-50%(3), 10-25%(2), 0-10%(1), N/A(0).
- **Question 4.** When GROOT correctly locates the root cause, do you find that the result "graph" provided by GROOT helps you understand how and why the incident happens?
- **Question 5.** When GROOT does not correctly locate the root cause, does the result "graph" make it easier for your investigation of the root cause?



(a) From 14 GROOT users (b) From 6 GROOT developers

Fig. 9: Survey results

Figure 9a shows the results of the user survey. We can see that most users find GROOT very useful to locate the root cause. The average score for Question 1 is 3.79, and 11 out of 14 participants find GROOT very helpful. As for Question 3, GROOT saves the triage time by 25-50%. Even in cases that GROOT cannot correctly locate the root cause, it is still helpful to provide information for further investigation with an average score of 3.43 in Question 5.

For the developer survey, we ask the 6 developers the following 5 questions (Questions 2-5 have the same choices as Question 1):

- **Question 1.** Overall, how convenient is it to change and customize events/rules/domains while using GROOT? Answer choices: Convenient(4), Somewhat Convenient(3), Not Convenient(2), Difficult(1).
- **Question 2.** How convenient is it to *change/customize event models* while using GROOT?
- **Question 3.** How convenient is it to *add new domains* while using GROOT?
- **Question 4.** How convenient is it to *change/customize causality rules* while using GROOT?
- **Question 5.** How convenient is it to change/customize GROOT compared to other SRE tools?

Figure 9b shows the results of the developer survey. Overall, most developers find it convenient to make changes on and customize events/rules/domains in GROOT.

B. *Lessons learned*

In this section, we share the lessons learned in terms of technology transfer and adoption on using GROOT in production environments.

[1]The GROOT researchers and developers who are authors of this paper are excluded.

Embedded in Practice. To build a successful RCA tool in practice, it is important to embed the R&D efforts in the live environment with SRE experts and users. We have a 30-minute routine meeting daily with an SRE team to manually test and review every site incident. In addition, we actively reach out to the end users for feedback. For example, the users found our initial UI hard to understand. Based on their suggestions, we have introduced alert enrichment with the detailed context of most events, raw metrics, and links to other tools for the next steps. We also make the UI interactive and build user guides, training videos, and sections. As a result, GROOT has become increasingly practical and well adopted in practice. We believe that R&D work on observability should be incubated and grown within daily SRE environments. It is also vital to bring developers with rich RCA experience into the R&D team.

Vertical Enhancements. High-confidence and automated vertical enhancements can empower great experiences. GROOT is enhanced and specialized in critical scenarios such as grouped related alerts across services or critical business domain issues, and large-scale scenarios such as infrastructure changes or database issues. Furthermore, the end-to-end automation is also built for integration and efficiency with anomaly detection, RCA, and notification. For notification, domain business anomalies and diagnostic results are sent through communication apps (e.g., slack and email) for better reachability and experience. Within 18 months of R&D, GROOT now supports 18 business domains and sub-domains of the company. On average, GROOT UI supports more than 50 active internal users, and the service sends thousands of results every month. Most of these usages are around the vertical enhancements.

Data and Tool Reliability. Reliability is critical to GROOT itself and requires a lot of attention and effort. For example, if a critical event is missing, GROOT may infer a totally different root cause, which would mislead users. We estimate the alert accuracy to be greater than 0.6 in order to be useful. Recall is even more important since GROOT can effectively eliminate false positive alerts based on the casual ranking. Since there are hundreds of different metrics supported in GROOT, we spend time to ensure a robust back end by adding partial and dynamic retry logic and high-efficiency cache. GROOT's unsuccessful cases can be caused by imperfect data, flawed algorithms, or simply code defects. To better trace the reason behind each unsuccessful case, we add a tracing component. Every GROOT request can be traced back to atomic actions such as retrieving data, data cleaning, and anomaly detection via algorithms.

Trade-off among Models. The accuracy and scalability trade-off among anomaly detection models should be carefully considered and tested. In general, some algorithms such as deep-learning-based or ensemble models are more adaptive and accurate than typical ones such as traditional ML or statistical models. However, the former requires more computation resources, operational efforts, and additional system complexities such as training or model fine-tuning. Due to the actual complexities and fast-evolving nature of our context, it is not possible to scale each model (e.g., deep-learning-based models), nor have it deeply customized for every metric at every level. Therefore, while selecting models, we must make careful trade-off in aspects such as accuracy, scalability, efficiency, effort, and robustness. In general, we first set different "acceptance" levels by analyzing each event's impact and frequency, and then test different models in staging and pick the one that is good enough. For example, a few alerts such as "high thread usage" are defined by thresholds and work just fine even without a model. Some alerts such as "service client error" are more stochastic and require coverage on every metric of every service, and thus we select fast and robust statistical models and actively conduct detection on the fly.

Phased Incorporation of ML. In the current industrial settings, ML-powered RCA products still require effective knowledge engineering. Due to the higher complexity and lower "signal to noise ratio" of real production incidents, many existing approaches cannot be applied in practice. We believe that the knowledge engineering capabilities can facilitate adoption of technologies such as AIOps. Therefore, GROOT is designed to be highly customizable and easy to infuse SRE knowledge and to achieve high effectiveness and efficiency. Moreover, a multi-scenario RCA tool requires various and interpretable events from different detection strategies. Auto-ML-based anomaly detection or unsupervised RCA for large service ecosystems is not yet ready in such context. As for the path of supervised learning, the training data is tricky to label and vulnerable to potential cognitive bias. Lastly, the end users often require complete understanding to fully adopt new solutions, because there is no guarantee of correctness. Many recent ML algorithms (e.g., ensemble and deep learning) lack interpretability. Via the knowledge engineering and graph capabilities, GROOT is able to explain diversity and causality between ML-model-driven and other types of events. Moving forward, we are building a white-box deep learning approach with causal graph algorithms where the causal link weights are parameters and derivable.

VII. CONCLUSION

In this paper, we have presented our work around root cause analysis (RCA) in industrial settings. To tackle three major RCA challenges (complexities of operation, system scale, and monitoring), we have proposed a novel event-graph-based approach named GROOT that constructs a real-time causality graph for allowing adaptive customization. GROOT can handle diversified anomalies and activities from the system under analysis and is extensible to different approaches of anomaly detection or RCA. We have integrated GROOT into eBay's large-scale distributed system containing more than **5,000** microservices. Our evaluation of GROOT on a data set consisting of 952 real production incidents shows that GROOT achieves high accuracy and efficiency across different scenarios and also largely outperforms baseline graph-based approaches. We also share the lessons learned from deploying and adopting GROOT in production environments.

References

[1] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables DevOps: Migration to a cloud-native architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.

[2] M. Solé, V. Muntés-Mulero, A. I. Rana, and G. Estrada, "Survey on models and techniques for root-cause analysis," *arXiv preprint arXiv:1701.08546*, 2017.

[3] N. Zhao, P. Jin, L. Wang, X. Yang, R. Liu, W. Zhang, K. Sui, and D. Pei, "Automatically and adaptively identifying severe alerts for online service systems," in *Proceedings of 2020 IEEE Conference on Computer Communications*. IEEE, 2020, pp. 2420–2429.

[4] J. Xu, Y. Wang, P. Chen, and P. Wang, "Lightweight and adaptive service api performance monitoring in highly dynamic cloud environment," in *Proceedings of 2017 IEEE International Conference on Services Computing*. IEEE, 2017, pp. 35–43.

[5] L. Tang, T. Li, F. Pinel, L. Shwartz, and G. Grabarnik, "Optimizing system monitoring configurations for non-actionable alerts," in *Proceedings of 2012 IEEE Network Operations and Management Symposium*. IEEE, 2012, pp. 34–42.

[6] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 74–89, 2003.

[7] H. Zawawy, K. Kontogiannis, and J. Mylopoulos, "Log filtering and interpretation for root cause analysis," in *Proceedings of 2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–5.

[8] V. Nair, A. Raul, S. Khanduja, V. Bahirwani, Q. Shao, S. Sellamanickam, S. Keerthi, S. Herbert, and S. Dhulipalla, "Learning a hierarchical monitoring system for detecting and diagnosing service issues," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2015, pp. 2029–2038.

[9] S. Lu, B. Rao, X. Wei, B. Tak, L. Wang, and L. Wang, "Log-based abnormal task detection and root cause analysis for Spark," in *Proceedings of 2017 IEEE International Conference on Web Services*. IEEE, 2017, pp. 389–396.

[10] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019, pp. 19–33.

[11] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," in *Proceedings of the 25th ACM Symposium on Operating Systems Principles*. ACM, 2015, pp. 378–393.

[12] H. Xu, W. Chen, N. Zhao, Z. Li, J. Bu, Z. Li, Y. Liu, Y. Zhao, D. Pei, Y. Feng *et al.*, "Unsupervised anomaly detection via variational auto-encoder for seasonal KPIs in web applications," in *Proceedings of the 2018 World Wide Web Conference*. ACM, 2018, pp. 187–196.

[13] M. Ma, W. Lin, D. Pan, and P. Wang, "Ms-rank: Multi-metric and self-adaptive root cause diagnosis for microservice applications," in *Proceedings of 2019 IEEE International Conference on Web Services*. IEEE, 2019, pp. 60–67.

[14] Y. Meng, S. Zhang, Y. Sun, R. Zhang, Z. Hu, Y. Zhang, C. Jia, Z. Wang, and D. Pei, "Localizing failure root causes in a microservice through causality inference," in *Proceedings of 2020 IEEE/ACM 28th International Symposium on Quality of Service*. IEEE, 2020, pp. 1–10.

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

[16] 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.

[17] 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 Very Large Data Base Endowment*, vol. 12, no. 12, pp. 1942–1945, 2019.

[18] H. Baek, A. Srivastava, and J. Van der Merwe, "Cloudsight: A tenant-oriented transparency framework for cross-layer cloud troubleshooting," in *Proceedings of 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2017, pp. 268–273.

[19] G. Da Cunha Rodrigues, R. N. Calheiros, V. T. Guimaraes, G. L. d. Santos, M. B. De Carvalho, L. Z. Granville, L. M. R. Tarouco, and R. Buyya, "Monitoring of cloud computing environments: Concepts, solutions, trends, and future directions," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. ACM, 2016, pp. 378–383.

[20] *How SRE teams are organized, and how to get started*, Accessed: 2020-12-10, https://cloud.google.com/blog/products/devops-sre/how-sre-teams-are-organized-and-how-to-get-started/.

[21] H. Nguyen, Z. Shen, Y. Tan, and X. Gu, "Fchain: Toward black-box online fault localization for cloud systems," in *Proceedings of 2013 IEEE 33rd International Conference on Distributed Computing Systems*. IEEE, 2013, pp. 21–30.

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

[23] M. Ma, Z. Yin, S. Zhang, S. Wang, C. Zheng, X. Jiang, H. Hu, C. Luo, Y. Li, N. Qiu *et al.*, "Diagnosing root causes of intermittent slow queries in cloud databases," *Proceedings of the Very Large Data Base Endowment*, vol. 13, no. 8, pp. 1176–1189, 2020.

[24] J. Schoenfisch, C. Meilicke, J. von Stülpnagel, J. Ortmann, and H. Stuckenschmidt, "Root cause analysis in it infrastructures using ontologies and abduction in markov logic networks," *Information Systems*, vol. 74, pp. 103–116, 2018.

[25] Á. Brandón, M. Solé, A. Huélamo, D. Solans, M. S. Pérez, and V. Muntés-Mulero, "Graph-based root cause analysis for service-oriented and microservice architectures," *Journal of Systems and Software*, vol. 159, p. 110432, 2020.

[26] D. Y. Yoon, N. Niu, and B. Mozafari, "Dbsherlock: A performance diagnostic tool for transactional databases," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 1599–1614.

[27] V. Jeyakumar, O. Madani, A. Parandeh, A. Kulshreshtha, W. Zeng, and N. Yadav, "Explainit!–a declarative root-cause analysis engine for time series data," in *Proceedings of the 2019 International Conference on Management of Data*. ACM, 2019, pp. 333–348.

[28] H. Jayathilaka, C. Krintz, and R. Wolski, "Performance monitoring and root cause analysis for cloud-hosted web applications," in *Proceedings of the 26th International Conference on World Wide Web*. ACM, 2017, pp. 469–478.

[29] M. A. Marvasti, A. V. Poghosyan, A. N. Harutyunyan, and N. M. Grigoryan, "An anomaly event correlation engine: Identifying root causes, bottlenecks, and black swans in IT environments," *VMware Technical Journal*, vol. 2, no. 1, pp. 35–45, 2013.

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

[31] J. Qiu, Q. Du, K. Yin, S.-L. Zhang, and C. Qian, "A causality mining and knowledge graph based method of root cause diagnosis for performance anomaly in cloud applications," *Applied Sciences*, vol. 10, no. 6, p. 2166, 2020.

[32] *Surus*, Accessed: 2020-08-15, https://github.com/Netflix/Surus.

[33] M. Ma, S. Zhang, D. Pei, X. Huang, and H. Dai, "Robust and rapid adaption for concept drift in software system anomaly detection," in *Proceedings of 2018 IEEE 29th International Symposium on Software Reliability Engineering*. IEEE, 2018, pp. 13–24.

[34] R. Bhagwan, R. Kumar, R. Ramjee, G. Varghese, S. Mohapatra, H. Manoharan, and P. Shah, "Adtributor: Revenue debugging in advertising systems," in *Proceedings of 11th USENIX Symposium on Networked Systems Design and Implementation*. USENIX, 2014, pp. 43–55.

[35] Y. Sun, Y. Zhao, Y. Su, D. Liu, X. Nie, Y. Meng, S. Cheng, D. Pei, S. Zhang, X. Qu *et al.*, "Hotspot: Anomaly localization for additive KPIs with multi-dimensional attributes," *IEEE Access*, vol. 6, pp. 10 909–10 923, 2018.

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

[37] C. Manning, P. Raghavan, and H. Schütze, "Introduction to information retrieval," *Natural Language Engineering*, vol. 16, no. 1, pp. 100–103, 2010.