2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

# CloudRanger: Root Cause Identification for Cloud Native Systems

Ping Wang *†‖, Jingmin Xu †‡, Meng Ma §, Weilan Lin†, Disheng Pan ¶, Yuan Wang ‡, Pengfei Chen ‡
* *National Engineering Research Center for Software Engineering, Peking University,*
† *School of Software and Microelectronics, Peking University,*
‡ *IBM Research China,*
§ *School of Electronics Engineering and Computer Science, Peking University,*
¶ *School of Electronic and Computer Engineering, Peking University Shenzhen Graduate School,*
‖ *Key Laboratory of High Confidence Software Technologies (PKU), Ministry of Education*
*§†¶‖ {pwang, mameng, weilan.lin, pdisheng}@pku.edu.cn; ‡{xujingm, crlwangy, cpfchen}@cn.ibm.com

***Abstract*—As more and more systems are migrating to cloud environment, the cloud native system becomes a trend. This paper presents the challenges and implications when diagnosing root causes for cloud native systems by analyzing some real incidents occurred in IBM Bluemix (a large commercial cloud). To tackle these challenges, we propose CloudRanger, a novel system dedicated for cloud native systems. To make our system more general, we propose a dynamic causal relationship analysis approach to construct impact graphs amongst applications without given the topology. A heuristic investigation algorithm based on second-order random walk is proposed to identify the culprit services which are responsible for cloud incidents. Experimental results in both simulation environment and IBM Bluemix platform show that CloudRanger outperforms some state-of-the-art approaches with a 10% improvement in accuracy. It offers a fast identification of culprit services when an anomaly occurs. Moreover, this system can be deployed rapidly and easily in multiple kinds of cloud native systems without any predefined knowledge.**

***Keywords*-Cloud native systems, Micro-service architecture, Root cause analysis, Anomaly detection, Causality**

## I. Introduction

Nowadays, increasingly business applications are "born on cloud" for taking advantages of all the benefits of a cloud native system such as horizontal design, continuous delivery, etc. [1]. The three properties "container packaged", "dynamically managed" and "microservice oriented" of a cloud native system allow easy abstraction and modularity for implementation, reuse, as well as independent component scaling [2]. The microservice oriented architecture distributing different responsibilities of the system into autonomous services enhances the cohesion and decreases the coupling, which allows the architecture of an individual service to evolve through continuous refactoring [3]. However, it can yield hundreds, or even thousands, of microservices, which introduces new challenges in managing both the deployment as well as the performance of cloud native systems.

First, monitoring microservice can be a big challenge due to the dynamic natures of the elastic environment. The number microservices can increase or decrease instantly in order to react to workload changes. Furthermore, a container can be deactivated, restarted, or even migrated to different nodes for system resiliency or resource optimization. Canary release [4], which is a technique used for reducing the risk of introducing new software versions, does both as frequent as new versions of a microservice are delivered. The dynamics make it difficult to track the containers in the context of their associated business transactions. The monitoring solution has to be able to pick up from right where it left off when a container is deactivated and subsequently restarted in order to provide continued historical performance of the container.

Second, the best strategy for managing the performance of a cloud native system is to measure both the performance of business transactions holistically as well as the individual microservices because a performance problem in an individual microservice may not surface at the business transaction level. However, the traditional approach to performance baselining can be a daunting undertaking for cloud native systems because you have to construct baselines for each business transaction and microservice. The sheer scale and magnitude of managing thousands of business transactions and microservices negates the practicality of manually instrumenting them for performance monitoring and setting realistic service level agreement (SLA) thresholds. The automation of such tasks is always challenged by middleware adaptivity and business transaction coverage. In addition, the baselines also have to catch up with the pace of the version progression of each microservice, but it is quite challenging due to the continuous delivery nature of the microservice world.

Third, compared to the traditional monolith, business transactions in a cloud native system usually have a much longer calling path with dozens of distributed microservices participating. Any performance problem or failure in the downstream nodes can be quickly propagated backwards to the upstream nodes, and eventually crashes the entire system if the problem is not identified and isolated in time. Some cloud design patterns, e.g. circuit breaker, can be used to mitigate the issue for improving the resiliency of the system, but it hinders the problem from being quickly identified from performance perspective.

In order to address these challenges, we propose Cloud

978-1-5386-5815-4/18/$31.00 ©2018 IEEE
DOI 10.1109/CCGRID.2018.00076

Ranger, a novel approach to performance management for cloud native systems. It basically takes four steps: anomaly detection, impact graph construction, correlation calculation, and root cause identification. It treats a cloud native system as a black box and monitors its performance in the service endpoints. Once an anomaly is detected, an impact graph is constructed based on dynamic causal relationship analysis against the observed performance metrics. A heuristic investigation algorithm based on second order random walk against the graph is then used to identify the problematic services. As you will see, this approach automatically constructs the impact graph with no need to trace and aggregate business transactions. Therefore, it is immune from the backend dynamics and performance baselining. We developed a prototype system and validated its effectiveness against a sample cloud native system running on top of IBM Bluemix Platform through comparing with selected state of the art approaches and tools. The results shows CloudRange gets 10% improvements in accuracy for root cause identification without given call graphs compared with some state-of-the-art.

The remainder of this paper is organized as follows. Related works are summarized in Section 2. Section 3 presents our monitoring architecture and dataset. Section 4 elaborates our proposed solution, CloudRanger, in terms of its high-level framework and details. Experiments and evaluations are included in Section 5. We discuss the advantages of CloudRanger and conclude the paper in Section 6.

## II. Related works

Significant research efforts have been devoted to related topics like finding root causes of anomalies in computer networks [5, 8-12]. In computer networks, the data is usually real-time metrics/events or link information between network devices [6, 7, 13], and a strong assumption is made that the links in the computer network structure represent a reliable dependency. This is true in computer network area. But for large-scale cloud native applications, this is not the case due to two facts. One is that the linkage between running nodes in cloud is usually loosely coupled and the switch from time to time (due to application scale out/in or load balancer). The other one is that some popular cloud design patterns such as circuit breaker, bulkhead, and so on can further change the error propagating pattern among cloud native services. Hence, in [14], Kim et. al., proposed a real-time metric collection system and anomaly detection framework named MonitorRank. It is an unsupervised and heuristic method based on random walking strategy. MonitorRank uses the historical and current performance metrics of each services as its input, along with the call graph generated between them. MonitorRank does not assume the reliable dependency between services, but it does need the call graph as the base. In cloud native environment, it costs much to capture the precise call graph. You need to deploy a whole set of topology capturing infrastructure such as Zipkin [15] to get a call graph in such a dynamic environment, which is usually impossible to perturb a running cloud platform. Another problem of MonitorRank is that even you have the precise call graph, you cannot assume the error propagating paths are always embedded in it. For example, due to the resource sharing nature of the cloud, several containers are running on the same physical node. One container's defect exhausts the network bandwidth and this will cause the problems of other containers. However, in call graph, these containers may not have network connections among them. In this paper, a data driven impact graph is proposed to replace the need of precise call graph to capture the error propagating pattern, a heuristic investigation algorithm based on second order random walk against the impact graph is used to handle the various cloud design patterns and identify the problematic services. Anomaly detection algorithms is also related to this paper, since an anomaly detector is needed to detector end user facing anomaly as the start point of our method, in this paper, we use our previous work [7] as the anomaly detection algorithm since it is more suitable to a dynamic workload environment and unsupervised comparing with other algorithms listed in [16]. Besides, traditional approaches did not consider about a higher order correlation when they infer the root causes leading to a biased result such as [17].

## III. Background and motivation

On IBM Bluemix platform, we are building and operating large-scale cloud native applications and services either for customer and platform itself, for example, IBM runs over 200 categories of public micro-services including Big Data, IBMWatson, Data Analytics, Internet of Things, Security, DevOps, Application, etc. [18], and each category of micro-service provides a bunch of service instances, APIs and runtimes. These services are running on thousands of machines in multiple cloud centers around the world, severing for over a million of users and producing over 1.5 billion API requests every single day according to a recent report [19]. In real production operations, the service deficiencies occasionally cause anomalies as high service latency and low usability, which propagates along the platform and has chance to crash the system finally. To keep the services running continuously, reliably, and diagnose anomalies quickly, heath checks and performance metric (e.g. latency and throughput) are usually collected for an application in a regular intervals as an indication of its basic performance status. In Table I, the details of collected dataset is summarized for a real anomaly in Bluemix system. In this incident, Bluemix web UI and command-line user interface (CLI) are slow in response time which is regarded as "performance down-gradation". Note that this dataset only records some key Bluemix system APIs and removes most unrelated application APIs. In Figure 1, performance metrics

are shown for 4 typical APIs selected from 1732 APIs. The main process of manual root cause identification is conducted as follows.

Table I
METRICS NOTATION

| Type | Details |
|---|---|
| **Duration** | 7200 seconds (2 hours) |
| **Interval** | every 1 second |
| **Total Metrics** | >11,000,000 (latency / throughput) |
| **Total API** | 1732 |
| **End User Facing API** | 54 |
| **Raw Data Size** | 4.6GB |

**Metrics investigation.** First, we examine different kind of metrics for their effectiveness. As Figure 1 shows, given the performance degradation anomaly, the latency is better to reflect the abnormal status of service compared to the throughput. When an anomaly occurs, the latency of Bluemix Dashboard API increases sharply however its throughput keeps relatively stable most of the times. It is because throughput is more dynamic as it is associated with the external API calling behavior (see Bluemix Console API for a clear instance). Besides, we find that all the original metrics are very sparse with 1-second sampling interval. Therefore, we need to aggregate them with an appropriate time interval. Meanwhile, the range of metrics varies for different APIs, for example. Nature Language Processing (NLP) service is time consuming so that its latency is determined by a specific task, sometimes thousand times longer than other services.

**Topology investigation.** The actual root cause in this incident is that several Bluemix event process components run out of memory. It propagates in the infrastructure and causes the console slowing down and finally freezes the dashboard. Accordingly, Figure 1 compares three kinds of services related to anomaly, where Event Component API stands for the root cause service, Bluemix Console API is the intermediate service in anomaly propagation, and Bluemix Dashboard API is the front-end service with anomaly. However, it is challenging to find this propagation pattern "Event $\rightarrow$ Console $\rightarrow$ Dashboard" from total 1732 different APIs. Worse still, some services are ambiguous to judge whether they participate in the anomaly propagation for the non-expert. For example, we find there are substantial increase in latency of NPL and IBM Watson API. While they are actually not related with the anomaly propagation in this incident. In real-world systems, the anomaly propagation path could be much more complex as it may affect hundreds of related services. In many situations, it is impossible to have complete knowledge about the relationships between all services. So, we need a methodology to establish the impact graph from raw metrics, helping us depict and analyze the anomaly propagation.

To keep this huge platform running continuously and reliably, there is a dedicated Bluemix SRE (Site Reliability Engineer) team insensitively watching and fixing anomalies. But due to the complexity of the system, it is usually difficult and time-consuming to find the actual root cause even for the experienced SRE. Without using any automation tools, an SRE needs to spend 3 hours on average in identifying the root causes of an incident. Therefore, our work in this paper aims to help SRE to quickly identify which service or component is most likely the one leading to the incident.

## IV. PROBLEM AND OUR SOLUTION

### A. Problem definition

In this work, we assume a cloud native system is composed by various services. And we treat it as a black box. It means we only have access to the basic performance metrics for each service and have no other system domain knowledge such as the actual calling graph among these services. This assumption comes from the real cloud platform Bluemix. There are more than 5000 services communicating and collaborating each other to support and control the platform. Each service is monitored by collecting some basic metrics like response time and throughput. But there are no way to get the precise calling graph among these 5000 services since they are so dynamic and complex. So this assumption makes the proposed method more general for various situations. Without loss of generality, for a certain kind of metric, we suppose that an anomaly is observed in a frontend service $v_{fe}$ in time period $[1, t]$ from service set $V$ (Other services are called "backend" correspondingly). Let an $n \times t$ matrix $\mathbf{T}$ where $[\mathbf{T}]_{i,j} = t_{i,j}$ be the metric data of node $v_i$ at time $j \in [1, t]$ for all the $n$ services node in $V$. Therefore, our goal is to find a set of services $v_{rc} \subset V$ , which cause the observed anomaly in $v_{fe}$ given $\mathbf{T}$. Table II lists the notations in this paper.

Table II
NOTATIONS

| Notation | Definitions |
|---|---|
| G(V, E) | directed graph $G$ with vertex and edge set $V, E$ |
| $v_{fe}, \mathbf{v}_{rc}$ | frontend service; set of root cause services |
| $I_i, O_i$ | set of in and out-neighbor nodes of node $i$ |
| $n, m, t$ | number of services; edges; metrics |
| $t_{i,j}$ | monitor metric of service node $i$ at time $j \in [1, t]$ |
| $\mathbf{t}_i, \mathbf{T}$ | $[\mathbf{t}_i]_j = t_{i,j}$; $\mathbf{T} : n \times t$ matrix, $[\mathbf{T}]_{i,j} = t_{i,j}$ |
| $\bigtriangledown_i$ | gradient matrix of $\mathbf{t}_i$, $[\bigtriangledown_i]_k = [\mathbf{t}_i]_k - [\mathbf{t}_i]_{k-1}$ |
| $c_{i,j}$ | correlation score of service $i$ to $j$ |
| $p_{i,j}$ | transition probability from service $i$ to $j$ |
| $p_{k,i,j}$ | transition prob. from $i$ to $j$ from $k$ |
| $p_{u,v}$ | transition prob. from edge $u$ to $v$ |
| $\mathbf{C}$ | $m \times m$ correlation matrix, $[\mathbf{C}]_{i,j} = c_{i,j}$ |
| $\mathbf{P}$ | node-to-node transition matrix, $[\mathbf{P}]_{i,j} = p_{i,j}$ |
| $\mathbf{M}$ | edge-to-edge transition matrix, $[\mathbf{M}]_{u,v} = p_{u,v}$ |
| $\sigma$ | threshold in anomaly detection |
| $\alpha$ | significance for conditional independence tests |
| $\beta, \rho$ | second-order, backward transition constant |
| $S(v_i, v_j)$ | set of nodes that d-separate $i$ and $j$ |

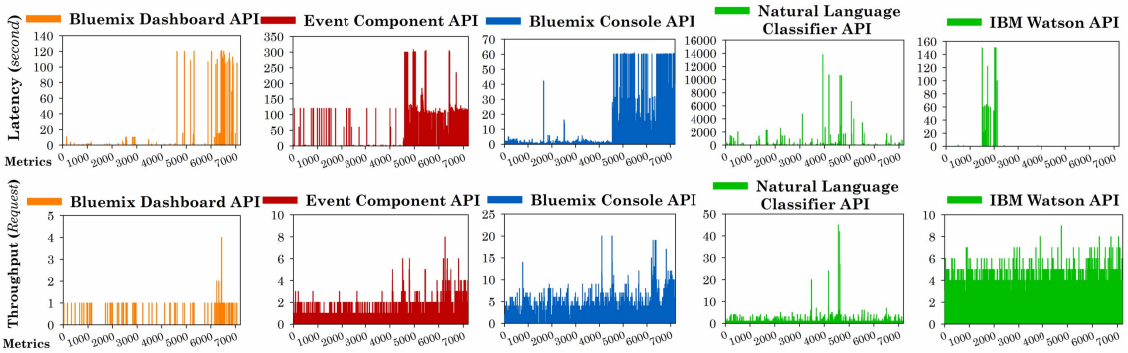


Figure 1. Performance monitoring metrics for key services

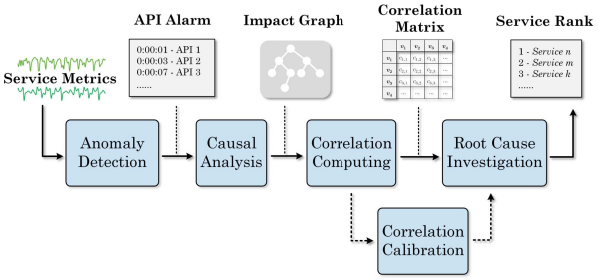


Figure 2. The framework of CloudRanger

### B. CloudRanger Framework

We propose a novel framework to solve this problem, named CloudRanger. In order to rank services according to their potentiality contributing to the given anomaly, CloudRanger decomposes the task into four stages: anomaly detection, impact graph construction, correlation computing/calibration, and root cause investigation (See Figure 2). In CloudRanger, we first detect the occurrence of an anomaly, to remove normal services and find abnormal candidates from numerous targets. Second, we extract a service impact graph by causal analysis based on intra-correlations between services. Third, we compute the correlation score for any pair of services as a quantitative measure of dependence between them with respect to given metrics. When needed, CloudRanger detects and calibrates special design patterns in micro-service architecture, to eliminate their negative effects on algorithm accuracy. Finally, we conduct a second-order random walk based on the impact graph and correlation scores, resulting into an anomalous service rank. We elaborate different phases of CloudRanger in the following subsections.

### C. Anomaly Detection

Anomaly detection is the starting phase of CloudRanger. In this paper, we leverage our previous work [7] as the anomaly detection method. There are two main benefits by using this method. First, it is a dynamic threshold method and it can follow and trace the dynamic running infrastructure and workload by introducing an on-line learning algorithm. Second, the method can individually detect all services' performance degradation by response time and throughput. Moreover, this will dramatically narrow down the root cause analysis scope since we can only focus on the services which are reported as the anomaly service.

When using the anomaly detection method in [7], there is an important factor, which will influence a lot in the later impact graph construction. That is he time window used to compute the average response time and throughput of a specific service. As the time window in anomaly detection will naturally be used as the aggregation window in the impact graph algorithm meaning that when two anomalies happened within the time window, we assume that they may possibly has directed impact relationship with each other. So how to select the time window is non-trivial in this paper. We determine it in this way. Since we aim to discover the impact relationship between services when an anomaly occurs, a reasonable selection of the aggregation window is the statistical average delay between services that call each other directly. In other words, they have one hop link on transaction call graph (direct call relationship). In practice, we calculate the statistics of the service average delay and do some experiments on the values around it. we choose several highly related services and do anomaly detection on them, then analysis if the anomalies of different services well synchronized with each other under different aggregation windows. The result shows that when the time window is small, for example, 1-second and 2-second, too many anomalies of different services are detected and they are not very synchronized with each other in time line. On the contrary, if we aggregate with a larger interval e.g., window = 10s, some anomalies will be ignored to catch the actual propagation topology even these anomalies should show good synchronized. We conclude from the experiments that 5 seconds is a preferred aggregation time window for Bluemix platform to group the anomalies.

### D. Impact Graph Construction

CloudRanger extracts the impact graph by causal analysis based on intra-correlations between services. We design the impact graph construction algorithm on the basis of PC-algorithm [20], which is under the assumption that the distribution of the observed metrics is faithful to a DAG (Directed Acyclic Graph). This assumption is true in our fine-grained monitoring architecture because API-level calling are naturally acyclic. We construct the skeleton without directions and generate a DAG according to the orientation rules mentioned in [20] which you will see in the following part. Different from previous approaches [21-23], our method is unsupervised, meaning that it does not need to label the normal and abnormal services manually.

**Impact graph**. Let $G(V, E)$ be a DAG describing the impact relationship between vertex (service) set $V$, it may contain undirected and directed edges, which have the following interpretation: (1) there is an edge between $v_i$ and $v_j$ if and only if these two services are conditionally dependent given all the possible subsets of $V \setminus \{v_i, v_j\}$; (2) a directed edge $v_i \rightarrow v_j$means that this directed edge is present in all DAGs. It stands for the change of $v_i$causes the change of $v_j$. In other word, $v_i$ impacts $v_j$; (3) an undirected edge $v_i - v_j$means that there is at least one DAG with edge $v_i \rightarrow v_j$and $v_i \leftarrow v_j$. Let $V$ be the input vertex set, given a significance level $\alpha \in (0, 1)$, the impact graph construction consists of four steps:

- **Step 1**. Generate a completely undirected graph on $V$;
- **Step 2**. Test conditional independence given subsets of adjacency sets at a given significance level $\alpha$ and delete edges if conditional independence is accepted;
- **Step 3**. Orient v-structures;
- **Step 4**. Orient remaining edges.

We leverage the concept of d-separation to determine the conditional independence, see [24] for details. Given a DAG model, two services $v_i$ and $v_j$ are d-separated by a set of services $S$, if and only if $v_i$ and $v_j$ are conditionally independent given any subset of $S$. We denote the set of services that d-separate $v_i$ and $v_j$ as a function $S(v_i, v_j)$. Let $G$ be the generated graph in Step 1 and $adj(G, v_i)$ denote the set of nodes in graph $G$ that are directly connected to $v_i$. In the second step, all pairs of services $(v_i, v_j)$ are tested for conditional independent given any single node in $adj(G, v_i) \setminus \{v_j\}$ or $adj(G, v_i) \setminus \{v_i\}$. If there is any service $v_k$ that makes $(v_i, v_j)$ conditionally independent, edge $v_i - v_j$ is removed and $v_k$ is inserted into $S(v_i, v_j)$ and $S(v_j, v_i)$. If all one-step adjacent pairs have been tested, a new graph is generated and the algorithm continues in this way by increasing the size of the conditioning set step by step. The process stops if all neighborhoods in the current graph are smaller than the size of the conditional set. The result of Step 2 is called skeleton in which every edge is still undirected. In this process, a higher significance level $\alpha$ place a more strict restriction on conditionally independent judgement. Therefore, the algorithm Step 2 removes less edge from $G$ and we obtain a result with more edges. Step 3 and 4 orient the skeleton into a DAG. First, we considers all triples $(v_i, v_j, v_k)$ that $v_i, v_k$ and $v_i, v_k$ are adjacent in skeleton however $v_i, v_j$ is not. All such triples are oriented as $v_i \rightarrow v_k \leftarrow v_j$ (also known as v-structure) if $v_k \notin S(v_i, v_j) \wedge v_k \notin S(v_j, v_i)$. For any remaining undirected edge, the algorithm checks whether any of its two possible directions introduces a new v-structure or a directed cycle. If so, that direction will be marked as invalid. Such edges can be found by repeatedly applying the following rules [25]:

- **Rule 1.** Orient $v_j - v_k$ into $v_j \rightarrow v_k$ whenever there is a directed edge $v_i \rightarrow v_j$ such that $v_i$ and $v_k$ are not adjacent (otherwise a new v-structure is created);
- **Rule 2.** Orient $v_i - v_j$ into $v_i \rightarrow v_j$ whenever there is a chain $v_i \rightarrow v_k \rightarrow v_j$ (otherwise a directed cycle is created);
- **Rule 3.** Orient $v_i - v_j$ into $v_i \rightarrow v_j$ whenever there are two chains $v_i - v_k \rightarrow v_j$ and $v_i - v_l \rightarrow v_j$ such that $v_k$ and $v_l$ are not adjacent (otherwise a new v-structure or a directed cycle is created).

For the sake of clarity, we summarize the process of impact graph construction in Algorithm 1. In order to facilitate the investigation of root causes, we reverse all the direction of edges in constructed DAG. The complexity of the Algorithm 1 is bounded by the largest degree in $G$. Let $k$ be the maximal degree of any vertex and let $n$ be the number of vertices. In the worst case, the number of conditional independence tests required by the algorithm is bounded by $2\binom{n}{k}\sum_{i=0}^{k}\binom{n-1}{i}$. Hence, our impact graph construction algorithm has a much lower complexity compared to the exponential complexity of Bayesian network-based algorithms [26].

**Example.** In Figure 3, we demonstrate the construction details with a demo impact graph based on the latency metrics of Bluemix Dashboard, Console, Event Component and Nature Language Classifier (namely NLP in the Figure). It starts with a complete undirected graph $G1$. First, we set $level = 0$ and test all service pairs for their conditional independence. For example, Event is independent with Dashboard in $G2$. Therefore, the edge ?Event-Dashboard? is removed from $G2$. Likely, we remove the edge between Dashboard and NLP in $G3$. When all the independences under the condition $level = 0$ are tested, we get $G4$. After that, we test the independences when $level = 1$ and $level = 2$. In $G5$, Event is conditionally independent with NLP given Dashboard. Therefore, $Dashboard \in S(Event, NLP)$. In $G7$, we find *Console* is conditionally independent with NLP given {*Event, Dashboard*}. Until $level = 3$, the algorithm stops because $|adj(G, v_i) \setminus \{v_j\}| < level, \forall v_i \in G$, so that we obtain the skeleton of the impact graph as shown in $G8$.

**Algorithm 1** Impact Graph Construction

**Input:** Vertex $V$, separation function $S$, significance $\alpha$
**Output:** Reversed DAG $G$

**new** DAG $G$ from $V$, $level = 0$
**for** $\forall(v_i, v_j) \in G$ **do**
  **if** $|adj(G, v_i) \setminus \{v_j\}| \geq level$ **then**
    **for** $\forall \mathbf{v_k} \subset adj(G, v_i) with |k| = level$ **do**
      **if** $v_i, v_j$ conditionally independent given $\mathbf{v_k}, \alpha$ **then**
        remove $v_i - v_j$ from $G$
        insert $\mathbf{v_k}$ into $S(v_i, v_j)$ and $S(v_j, v_i$
      **end if**
    **end for**
  **end if**
  $level \leftarrow level + 1$
**end for**
**for** $\forall v_i - v_j - v_k \in G, v_k \notin S(v_i, v_j) \wedge v_k \notin S(v_j, v_i)$ **do**
  replace $v_i - v_j - v_k$ by $v_i \rightarrow v_k \rightarrow v_j$
**end for**
**for** $\forall(v_i, v_j) \in G$ **do**
  check $v_i \rightarrow v_j$ and $v_i \leftarrow v_j$ with **Rule 1-3**
**end for**
**for** $\forall(v_i, v_j) \in G$ **do**
  replace $v_i \rightarrow v_j$ by $v_i \leftarrow v_j$
**end for**

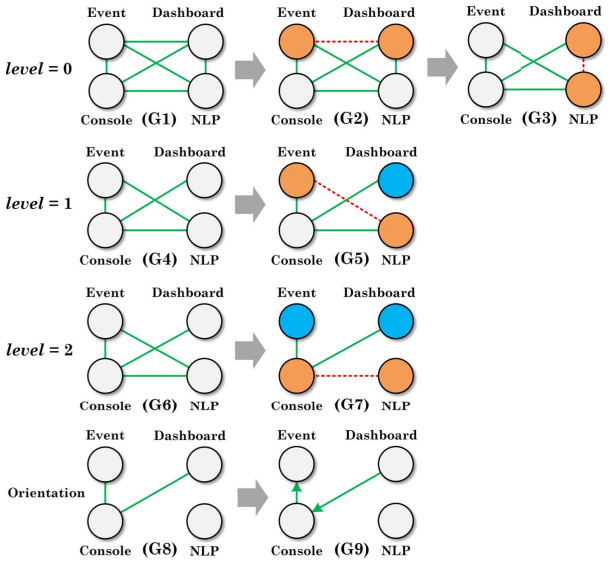


Figure 3. An example of impact graph construction

Finally, we orient $G8$ by Rule 1-3 and output the reversed graph shown in $G9$ representing the impact relationship $Event \rightarrow Console \rightarrow Dashboard$.

### E. Correlation calculation

Intuitively, one simple way of identifying the root cause possibility of a backend service is computing how its metric pattern is similar to the observation in the frontend service. This simple idea is basically the same as how we investigate the root cause manually, that is, if two services share similar abnormal patterns in a given metric series, they should be affected by the same root cause. Therefore, we define a correlation function to measure the similarity. Given a service set $V$ and its metric data matrix $\mathbf{T}$, for any pair of service $v_i$, $v_j \in V$, the correlation $c_{i,j}$ scores the relevance of $v_i$ to $v_j$ in terms of $\mathbf{T}$. The value of $c_{i,j}$ is in $[0, 1]$. A value of $c_{i,j} = 1$ implies completely correlation, whereas $c_{i,j} = 0$ implies that there is no correlation between them. We propose a revised Pearson correlation function as a measure of the linear dependence between two metric series for service $v_i$ and $v_j$, defined as

$$[C]_{i,j} = \left| \frac{\sum_{k=1}^{t}([t_i]_k - \bar{t}_j)([t_j]_k - \bar{t}_j)}{\sum_{k=1}^{t}\sqrt{([t_i]_k - \bar{t}_j)^2}\sqrt{([t_j]_k - \bar{t}_j)^2}} \right| \quad (1)$$

In particular, the correlation function used in this paper is actually the absolute values of original Pearson algorithm because both the positive and negative correlation exist in the anomaly propagation in cloud native systems. According to specific problem scenarios, different correlation functions can be defined in CloudRanger framework.

### F. Root Cause Identification

In the constructed impact graph, each node $v_i \in V$ indicates a service, each edge $e_{i,j} \in E$ is set to 1 when service $v_i$ is impacted by $v_j$. For simplicity, we use the notation of original correlation score $c_{i,j}$ in the following discussion. The algorithm proposed to find the root cause is inspired by the real-world system operator's behavior in manual root cause investigation. Assuming that we have no domain knowledge about the anomaly but only have the discovered impact graph $G$ and correlation score $\mathbf{C}$, one of the natural diagnosis methods is to randomly traverse services following $G$ with preferentially looking for a high correlation score $c_{i,j}$ regard to anomaly in frontend service $v_{fe}$. The traverse should not only consider the correlation of current service, but also take into account the relation between previous walked service. For example, during random walk, when the correlation of previous and current node is both high, it is more possible to find the root cause if we move forward instead of backward troubleshooting. To this end, we propose our scheme leveraging the concept of a second-order random walk [27]. In a first-order random walk, a surfer walks from service $v_i$ to $v_j$ with probability $p_{i,j}$. Let $X_t$ be the service visited by the surfer at time $t$, $p_{i,j}$ can be represented as conditional probability $P[X_t = v_j | X_{t-1} = v_i]$. The surfer visit $v_j$ proportionally to its correlation score to $v_{fe}$, i.e.,

$c_{j,fe}$. Hence, the node-to-node transition probability matrix $\mathbf{P}$ is defined as $[\mathbf{P}]_{i,j} = p_{i,j} = \frac{c_{j,fe}}{\sum_{l \in O_i} c_{l,fe}}$.

**Forward transition.** In our scheme, the surfer considers the previous visited node. Let $p_{k,i,j}$ be the transition probability from service $v_i$ to $v_j \in O_i$ given that $v_k \in I_i$ is previously visited, $p_{k,i,j} = P[X_{t+1} = v_j | X_{t-1} = v_k, X_t = v_i] = P[X_{t+1} = v_j, X_t = v_i | X_{t-1} = v_k, X_t = v_i]$. We consider edge-to-edge transitions, $p_{k,i,j}$, which can be denoted as $p_{k,i,j} = p_{u,v}$ if we let $u = (v_k, v_i)$ and $v = (v_i, v_j)$ be the edge from $v_k$ to $v_i$ and $v_i$ to $v_j$ respectively. In edge-to-edge transitions, we define an autoregressive model where $p^{'}_{k,i,j} = (1-\beta)p_{k,i} + \beta p_{i,j}$, taking the previously visited node into consideration. The parameter $\beta \in (0, 1]$ controls the strength of effect from the previous visited services. When $\beta = 1$, we have $p^{'}_{k,i,j} = p_{i,j}$, i.e., the edge-to-edge transition degenerates to node-to-node. Let $\mathbf{M}$ be an $m \times m$ transition matrix, with element $p_{u,v}$. Therefore, our forward transition probability calculates as:

$$[\mathbf{M}] = p_{k,i,j} = \frac{p^{'}_{k,i,j}}{\sum_{l \in O_i} p^{'}_{k,i,l}} = \frac{(1-\beta)p_{k,i} + \beta p_{i,j}}{\sum_{l \in O_i}[(1-\beta)p_{k,i} + \beta p_{i,l}]}$$

**Backward transition.** Given only forward transition, the surfer can only move forward along the impact graph even if the current service shows a high correlation with the frontend service while all other neighbor services do not. Correspondingly, we setup two additional types of transitions, namely *backward* and *selfward*, helping the surfer to find more routes and making its random walk more heuristic. Assume the surfer is visiting a service $v_i$ with a low correlation score $c_{i,fe}$. If all its out-neighbor services in $O_i$ show less correlation to the given anomaly, there will be no way to escape but trapped in this wrong route. Backward transition is introduced to resolve this issue. Let $p^b_{k,i,j}$ be the backward transition probability from current service $v_i$ to its in-neighbor $v_j \in I_i$ given $v_k$ previously visited, $p^b_{k,i,j}$ is restricted by a backward constant $\rho \in [0, 1)$, defined as

$$p^b_{k,i,j} = \rho \frac{p^{'}_{k,i,j}}{\sum_{l \in I_i} p^{'}_{k,i,l}} = \rho \frac{(1-\beta)p_{k,i} + \beta p_{i,j}}{\sum_{l \in I_i}[(1-\beta)p_{k,i} + \beta p_{i,l}]}$$

. The backward constant $\rho$ represent the restriction strength of the direction of impact graph on the surfer. If we set $\rho$ lower, the surfer is more restricted to the direction of the impact graph. Conversely, if the value of $\rho$ is higher, the surfer walks with more flexibility as it would explore downstream and upstream.

**Selfward transition.** The surfer is also encouraged to stay longer on the visiting service if none of its in and out-neighbor services are of high correlation score. To this end, we introduce selfward transition. Let $p^s_i$ be the selfward transition probability when $v_i$ is the visiting service. We denoted it as $p^s_{k,i,i}$ given $v_k$ previously visited. Specifically, $p^s_i$ is determined by the difference of $p_{k,i,i}$ and the maximum transition probability of its in and out-neighbor services. Hence, $p^s_{k,i,i} = max(0, p_{k,i,i} - \max_{l \in I_i \cup O_i} p_{k,i,l})$.

**Random walk algorithm.** Given $G(V, E)$ and $\mathbf{C}$, the surfer starts from $v_{fe}$, calculates the probability of forward, backward and selfward transition, and randomly walks following the impact graph. Different services are visited in sequence by randomly choosing the next service among their neighbors. We record that how many times for each service being visited and output the list descending as the root cause identification result. We summarize this process in the following algorithm. In CloudRanger framework, the route of random walk is determined by the impact relationship between services reflected by metrics. Hence, the constructed impact graph is more accurate than actual calling topology in specific situation when the interested anomaly occurs and propagates. Meanwhile, the impact graph contains implicit relationships which are not defined in the calling topology. Therefore, for a given anomaly, the result of random walk in CloudRanger framework is more heuristic, and more accurate.

**Algorithm 2** Random Walk

**Input:** $G(V, E), \mathbf{M}, v_{fe}$.
**Output:** $R[n]$.

```
new Array R[n], v_s = v_fe, v_p = v_fe
repeat
    v_p ← v_s
    for each l ∈ O_s calculate p_{p,s,l}
    for each l ∈ I_s calculate p^b_{p,s,l}
    calculate p^s_{p,s,s}, row normalize [M]_{p,s}
    v_s ← randomly choose from O_s ∪ I_s ∪ {v_s}
    R[s] ← R[s] + 1
until n rounds
R[n] ← Sort(R[n])
```

## V. Empirical Study

### A. Testbed and Evaluation Metric

We use both simulated environment and real-world production cases from Bluemix for validation. As the acurate service calling topology is hard to obtain in real complex production environment, in order to compare the algorithm accuracy with other methods, we use Pymicro [28] to simulate the micro-service-based systems and their anomaly propagation. Bluemix is our real production testbed. We use dozens of different Bluemix incidents to validate our method. For each incident, the typical data size comprises more than 10,000,000 metric points including response time and throughput collected for over 1,000 micro-service APIs during 2 hours which embraces the period when the incident happened.

Three kinds of methods are selected for comparison, namely random selection, TBAC [29] (Timing Behavior

Anomaly Correlation) and MonitorRank [14]. Random selection is a most basic way to investigate the root cause in random order given no domain knowledge about anomaly. The comparison between random selection and other three algorithms reflects the improvement by introducing metric-based knowledge into problem solving. TBAC is a non-heuristic method, working on the anomaly metric correlation and improved using a series of weighted rating based on the dependency relation graph [29]. We compare TBAC with heuristic algorithms to show the advantage and robustness of random walking in solving this issue. MonitorRank is a heuristic algorithm based on first-order random walk. MonitorRank requires to obtain the true calling topology of target system [14]. Therefore, the comparison between MonitorRank and CloudRanger first validate the advantage of using impact graph instead of following true service calling topology.

To quantify the performance of each algorithm, we introduce a performance metric for result accuracy. This metric indicates the probability that top $k$ result given by each algorithm includes the real root causes for all given anomaly cases, denoted as $AC@k$. A higher result of $AC@k$, especially when the value of $k$ is small, represents the algorithm identifies the actual root cause more accurate, resulting in less services for further investigation. Let $R[n]$ be the result rank of each service, specifically, $AC@k$ is defined on a set of given anomalies $A$ as:

$$AC@k = \frac{1}{|A|} \sum_{a \in A} \frac{\sum_{i<k} R[i] \in v_{rc}}{min(k, |v_{rc}|)}. \quad (2)$$

We also compare the overall performance of each algorithm by computing the average $AC@k$, which defines as:

$$Avg@k = \frac{1}{k|A|} \sum_{a \in A} \sum_{1 \le j \le k} AC@j. \quad (3)$$

Our evaluation system is implemented by Python. The impact graph construction algorithm is implemented with the R language. Experiments were performed on a workstation with an Intel Xeon CPU 3.4GHz and 64GB RAM running Java HotSpot Server VM in 64 bit Windows Server. All experimental results are averaged over twenty different rounds of test.

### B. Simulation Experiment and Analysis

***Impact graph construction.*** The first experiment examines the constructed impact graph based on the collected latency and throughput metrics from Pymicro system. Note that these collected metric data includes normal and abnormal observations. We record the latency when every time of calling for each service. In order to simulate the occurrence and propagation of anomalies, we randomly select several services and inject fault to them, for example, shutdown the service host or attacks using denial of service. Given $\omega = 5$ (every 5 seconds) and $\alpha = 0.1$, we calculated

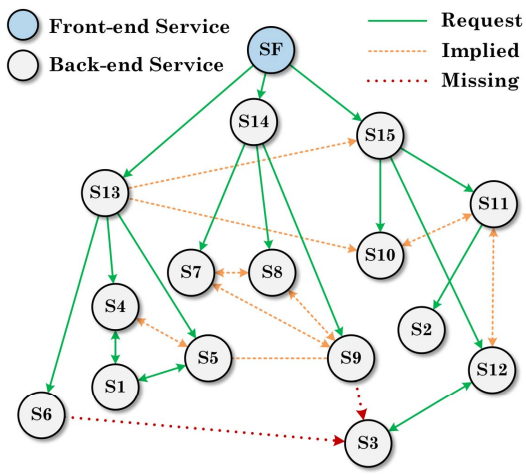


Figure 4. Pymicro impact graph

$t = 1500$ metric samples using Algorithm 1 to obtain the results shown in Figure 4. In this topology, "Request" means that edge exists in the actual calling topology of Pymicro. If the edge in the impact graph does not exist in the actual topology, we call it "implied". Conversely, if any linkage in the actual topology has not been discovered in impact graph, it is called "missing". We find that the constructed impact graph skeleton and directions are generally consistent and can reflect the actual service calling relationship in Pymicro. The implied edges in Figure 4 indicate that there are implicit associations between services. These edges are most bi-directional, for example, $S_4 \leftrightarrow S_5$ and $S_7 \leftrightarrow S_8 \leftrightarrow S_9$ as they call the same DB services in Pymicro. We note some edges are missing in the constructed impact graph compared with the actual calling topology, which implies that the missing calling relationship has very low relevancy with the given anomaly.

***Root cause identification.*** The second experiment compares the accuracy of selected algorithms. We use actual calling topology of Pymicro in TBAC and MonitorRank algorithm. For CloudRanger, we use the constructed impact graph. We compare the $AC@1, 3, 5$ and $Avg@5$ when $t = 1500$ based on latency and throughput respectively. The experimental result is summarized in Table III (we denote four algorithms as RD, TB, MR, SR for short). It shows that CloudRanger outperforms other algorithms in terms of accuracy with both latency and throughput metric. Especailly, CloudRanger provides an accuracy of 59.4% in the first recommended result and 85.2% for average top 5 result. In terms of the $Avg@5$, TBAC and MonitorRank have the accuracy of 47.0% and 73.7% respectively. We present an example of root cause result in Table IV, which shows the top 13 results given by four algorithms ( $S_1$ is the root cause, marked in bold). CloudRanger lists the accurate root cause at the first one of its result list, which appears

in the fourth or even later in other algorithms. Therefore, CloudRanger significantly improves the efficiency of root cause investigation in cloud native systems. However, we also find that the result based on throughput is not as good as the result based on latency. Because Pymicro is a system with single frontend service and synchronous request. As a result, the correlation scores based on throughput metrics are similar and hard to distinguish the pattern related with given anomalies.

Table III
ROOT CAUSE IDENTIFICATION RESULT

| | RD | TB | MR | SR |
|---|---|---|---|---|
| **Latency** | | | | |
| **AC@1** | 06.1% | 23.1% | 25.4% | **59.4%** |
| **AC@3** | 19.0% | 45.3% | 87.4% | **89.5%** |
| **AC@5** | 31.8% | 61.3% | 89.7% | **93.3%** |
| **Avg@5** | 30.1% | 47.0% | 73.7% | **85.2%** |
| **Throughput** | | | | |
| **AC@1** | 06.0% | 16.2% | 41.9% | **40.1%** |
| **AC@3** | 18.3% | 35.9% | 66.3% | **68.2%** |
| **AC@5** | 30.5% | 40.1% | 72.1% | **73.4%** |
| **Avg@5** | 31.8% | 43.7% | 64.1% | **68.8%** |

Table IV
TOP 13 RESULT OF PYMICRO

| RD | TB | MR | SR |
|---|---|---|---|
| Service 3 | Service 4 | Service 2 | **Service 1** |
| Service 5 | Service 13 | Service 3 | Service 3 |
| Service 13 | Service 10 | Service 5 | Service 2 |
| Service 11 | Service 9 | **Service 1** | Service 4 |
| Service 4 | Service 6 | Service 6 | Service 11 |
| Service 8 | Service 15 | Service 15 | Service 8 |
| Service 12 | Service 2 | Service 7 | Service 6 |
| Service 10 | **Service 1** | Service 11 | Service 12 |
| Service15 | Service 14 | Service 4 | Service 5 |
| Service 7 | Service 7 | Service 12 | Service 9 |
| Service 9 | Service 11 | Service 13 | Service 13 |
| **Service 1** | Service 12 | Service 8 | Service 15 |
| Service 6 | Service 8 | Service 9 | Service 14 |

***Algorithm parameters.*** In the third experiment, we aim to evaluate the result accuracy with different parameters more specifically, $t$ and $\alpha$. We compare the accuracy of CloudRanger algorithm when increasing $t$ from 800 to 1500 and $\alpha$ from 0.01 to 0.5. According to our previous analysis, when $\alpha$ is larger, the algorithm is more likely to find potential impact relations. Likewise, in terms of $t$, the more metric input (means $t$ is higher), the result more accurately reflects the impact relationship between services. This is proved by the experimental result shown in Figure 5. It is worth noting that a small parameter $t$ results in a significant impact on the accuracy, $AC@1 < 20\%$ when $t = 800$ as indicated by the statistics. On the other hand, when the value $\alpha$ is too small, for example $\alpha = 0.05$. The $AC@1$ of CloudRanger is $66.3\%$, owing to the incomplete impact graph. The algorithm accuracy is significantly decreased as compared to the accuracy $91.1\%$ if we set $\alpha = 0.5$.

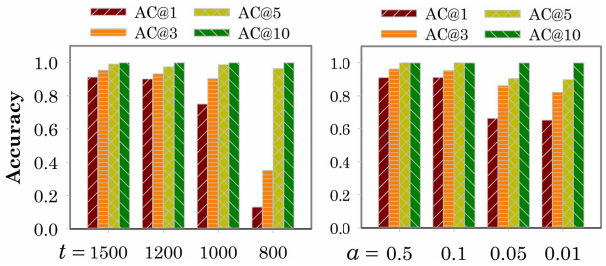


Figure 5. Accuracy of CloudRanger with different $t$ and $\alpha$

### C. Real Production Environment Validation

**Impact graph construction.** Due to the limited space, we select one of the aforementioned Bluemix incidents and present the constructed impact graph in this experiment. Taking into account that Bluemix is a very complex system, a large part of its services are irrelevant with a particular anomaly. In order to reduce the complexity and improve the efficiency of CloudRanger, we generate a root cause candidate list by choosing the top 5% alarmed service API by anomaly detection. In the experiment, we set $t = 1440, \alpha = 0.5, \omega = 5$. As shown in Figure 6, most of the abnormal services are connected by constructed impact graph, meaning they probably participate the anomaly propagation path. Note that service API $18 \leftarrow 13 \leftarrow 6$ represents the impact relationships $Event \leftarrow Console \leftarrow Dashboard$ we find in Figure 3. Some services (See service No. 1, 12 and 27) are isolated of the graph, so that they will not be taken into account in the following random walk. We are curious about this result and seek advice to IBM Bluemix SRE team. They find that these services represent HTML5 based WebSocket server and Watson NLP. Similar to our analysis in Figure 4, they are long connections and their response times make no sense. In other words, these services have no calling relationships with other services at all. Hence, CloudRanger can filter out some irrelevant services in the graph construction phase.

**Root cause identification.** To start the experiment, there are two front-end candidates in Figure 6, namely Service No.18 and No.26, which represent the end points of Bluemix Dashboard and Interface respectively. According to the judgement of SRE, the root cause should be Service 31, 30, 28 and 6 (marked underlined in the Figure). These service APIs are provided by event process components. If we cans identify these services as the root causes, SRE can quickly check the event process components and find the actual problem. Considering that the result based on latency is significantly better than throughput, and in view that the throughput will be more affected by the external calling behavior, the following experimental results are obtained by the metric of latency as it better reflects the characteristics of service.

**Algorithm parameters.** We evaluate the result accuracy

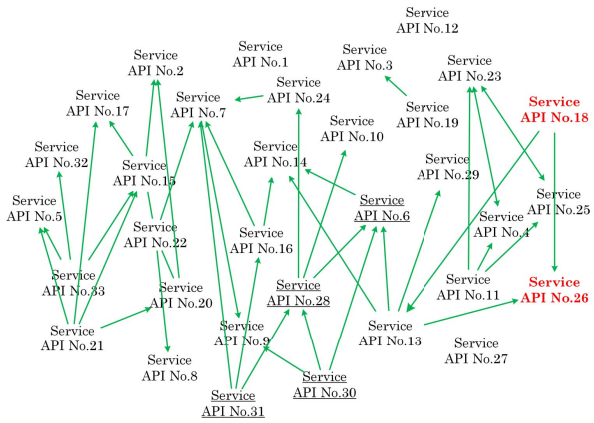


Figure 6. Constructed impact graph of Bluemix incident

with Bluemix incidents by setting $\omega = 1$, 2, 10, and 20 seconds. Consistent with our analysis in Section 4.3-Sampling interval, as the result shows (See Table V and VI), 2-sigma of average time delay outputs the best result, over 90% specifically and all the root causes are listed in front of the result in this incident example. Hence, it is a reasonable selection of the parameter $\omega$ is the statistical average time delay between services that call each other directly. If $\omega$ is too large, there will be overmuch edges that do not reflect the real calling relationship, and if $\omega$ is too small, some edges will be lost to catch the actual topology. We also compare the result with different values of $t$ and $\alpha$ in Bluemix production environment. Figure 7 shows the experimental result. Similar to the result in Pymicro system, we have a higher result accuracy with larger value of $t$ and $\alpha$.

Table V
ACCURACY WITH DIFFERENT $\omega$

| | $\omega = 1$ | $\omega = 2$ | $\omega = 5$ | $\omega = 10$ | $\omega = 20$ |
|---|---|---|---|---|---|
| $AC@1$ | 26.1% | 84.9% | **98.6%** | 90.3% | 12.0% |
| $AC@3$ | 36.5% | 69.0% | **92.5%** | 83.6% | 18.6% |
| $AC@5$ | 62.1% | 63.2% | **90.5%** | 59.3% | 24.6% |
| $Avg@5$ | 55.4% | 79.3% | **95.4%** | 82.9% | 35.3% |

Table VI
TOP 13 RESULT OF BLUEMIX INCIDENT

| $\omega = 1$ | $\omega = 2$ | $\omega = 5$ | $\omega = 10$ | $\omega = 20$ |
|---|---|---|---|---|
| $Service18$ | **Service 31** | **Service 31** | **Service 31** | $Service5$ |
| **Service 30** | **Service 30** | **Service 30** | **Service 6** | $Service20$ |
| **Service 31** | $Service24$ | **Service 28** | **Service 30** | $Service22$ |
| **Service 28** | **Service 28** | **Service 6** | $Service26$ | $Service21$ |
| $Service3$ | $Service9$ | $Service13$ | $Service9$ | **Service 30** |
| $Service22$ | $Service16$ | $Service26$ | $Service24$ | **Service 28** |
| $Service16$ | **Service 6** | $Service9$ | $Service17$ | **Service 31** |
| $Service24$ | $Service14$ | $Service16$ | $Service21$ | $Service7$ |
| $Service26$ | $Service18$ | $Service24$ | **Service 28** | $Service2$ |
| **Service 6** | $Service32$ | $Service7$ | $Service20$ | **Service 6** |
| $Service29$ | $Service33$ | $Service4$ | $Service16$ | $Service15$ |
| $Service32$ | $Service13$ | $Service20$ | $Service5$ | $Service24$ |
| $Service25$ | $Service8$ | $Service18$ | $Service32$ | $Service1$ |

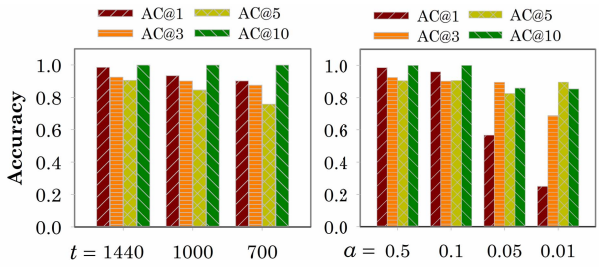


Figure 7. Accuracy in different $t$ and $\alpha$

## VI. CONCLUSION

This paper introduces CloudRanger, a novel approach for root cause identification in cloud native systems. Experimental results show that CloudRanger outperforms other methods in terms of accuracy and offers a fast identification of root causes. Advantages of CloudRanger over state-of-art methods are manifolds: i) we treat the cloud native system as a "black box" and make no assumption of predefined knowledge about the system. Therefore, CloudRanger can be applied into extensive application scenarios since it does not require a predefined service topology; ii) we can easily integrate domain knowledge and maintenance experiences into CloudRanger by replacing its correlation function or predefining the topology. For example, we can replace the correlation score function by the similarity of current metric and historical observations, which can better reflect the healthy states of IO-intensive or CPU-intensive services; iii) CloudRanger is a framework with high adaptability, which can be extended to handle more scenarios in cloud native systems. Without significant modifications, CloudRanger can also diagnose and analyze other complex networks such as sensor, social, and biological networks.

## ACKNOWLEDGMENT

The National Key R&D Program of China (2017YFB1200700), National Natural Science Foundation of China (61701007), China Postdoctoral Science Foundation (2016M600865) and IBM Shared University Research Project support this work. Meng ma is the corresponding author.

## REFERENCES

[1] Cloud Native Computing Foundation, http://www.nltk.org/, [Accessed on Nov 17, 2017].

[2] S. Newman, "Building Microservices", *O'Reilly*, 2015

[3] T. Erl, "Service-oriented architecture: concepts, technology, and design", *Pearson Education India*, 2005.

[4] Canary Release, https://martinfowler.com/bliki/CanaryRelease.html, [Accessed on Nov 17, 2017].

[5] M. lgorzata Steinder, and A. S. Sethi, "A survey of fault localization techniques in computer networks", *Science of computer programming*, vol. 53, no. 2, pp. 165-194, 2004.

[6] A. Homer, J. Sharp, L. Brader, M. Narumoto, and T. Swanson, "Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications", *Microsoft patterns & practices*, 2014.

[7] 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 (SCC)*, pp. 35-43, 2017.

[8] T. Ahmed, B. Oreshkin, and M. Coates, "Machine learning approaches to network anomaly detection", *in Proceedings of the 2nd USENIX workshop on Tackling computer systems problems with machine learning techniques*, pp. 1-6, 2007.

[9] Y. Liu, L. Zhang, and Y. Guan, "A distributed data streaming algorithm for network-wide traffic anomaly detection", *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 2, pp. 81-82, 2009.

[10] R. Jiang, H. Fei, and J. Huan, "Anomaly localization for network data streams with graph joint sparse PCA", *in Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 886-894, 2011.

[11] J. Gao, G. Jiang, H. Chen, and J. Han, "Modeling probabilistic measurement correlations for problem determination in large-scale distributed systems", *in 29th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pp. 623-630, 2009.

[12] C. Wang et al., "VScope: middleware for troubleshooting time-sensitive data center applications", *in ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pp. 121-141, 2012.

[13] L. Akoglu, H. Tong, and D. Koutra, "Graph based anomaly detection and description: a survey", *Data Mining and Knowledge Discovery*, vol. 29, no. 3, pp. 626-688, 2015.

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

[15] Zipkin, http://zipkin.io/, [Accessed on Nov 17, 2017].

[16] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey", *ACM Comput. Surv., 41(3)*, 2009.

[17] Y. Wu, Y. Bian, and X. Zhang, "Remember where you came from: on the second-order random walk based proximity measures", *Proc. VLDB Endow.*, vol. 10, no. 1, pp. 13-24, 2016.

[18] Cloud Connect China 2016, Sep 2016, Shanghai, China.

[19] Hystrix, https://github.com/Netflix/hystrix, [Accessed on Nov 17, 2017].

[20] P. Spirtes, C. N. Glymour, and R. Scheines, "Causation, prediction, and search", *MIT press*, 2000.

[21] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons, "Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control", *in Proceedings of OSDI*, vol. 4, pp. 16-16, 2004.

[22] S. Zhang, I. Cohen, M. Goldszmidt, J. Symons, and A. Fox, "Ensembles of models for automated diagnosis of system performance problems", *in Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pp. 644-653, 2005.

[23] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, "Capturing, indexing, clustering, and retrieving system history", *in ACM SIGOPS Operating Systems Review*, vol. 39, no. 5, pp. 105-118, 2005.

[24] D. Geiger, T. S. Verma, and J. Pearl, "d-separation: From theorems to algorithms", *arXiv preprint:1304. 1505*, 2013.

[25] M. Kalisch and P. Bhlmann, "Estimating high-dimensional directed acyclic graphs with the PC-algorithm", *Journal of Machine Learning Research*, vol. 8, no. 3, pp. 613-636, 2007.

[26] B. Ellis and W. H. Wong, "Learning causal Bayesian network structures from experimental data", *Journal of the American Statistical Association*, vol. 103, no. 482, pp. 778-789, 2008.

[27] Y. Wu, Y. Bian, and X. zhang, "Remember where you came from: on the second-order random walk based proximity measures", *Proc. VLDB Endow.*, vol. 10, no. 1, pp. 13-24, 2016.

[28] Pymicro, https://github.com/rshriram/pymicro, [Accessed on Nov 17, 2017].

[29] N. Marwede, M. Rohr, A. van Hoorn, and W. Hasselbring, "Automatic failure diagnosis support in distributed large-scale software systems based on timing behavior anomaly correlation", *in 13th European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 47-58, 2009.