Root Cause Detection in a Service-Oriented Architecture

Myunghwan Kim*
Stanford University
Stanford, CA, USA
mykim@stanford.edu

Roshan Sumbaly
LinkedIn Corporation
Mountain View, CA, USA
rsumbaly@linkedin.com

Sam Shah
LinkedIn Corporation
Mountain View, CA, USA
samshah@linkedin.com

ABSTRACT

Large-scale websites are predominantly built as a service-oriented architecture. Here, services are specialized for a certain task, run on multiple machines, and communicate with each other to serve a user's request. An anomalous change in a metric of one service can propagate to other services during this communication, resulting in overall degradation of the request. As any such degradation is revenue impacting, maintaining correct functionality is of paramount concern: it is important to find the root cause of any anomaly as quickly as possible. This is challenging because there are numerous metrics or sensors for a given service, and a modern website is usually composed of hundreds of services running on thousands of machines in multiple data centers.

This paper introduces MonitorRank, an algorithm that can reduce the time, domain knowledge, and human effort required to find the root causes of anomalies in such service-oriented architectures. In the event of an anomaly, MonitorRank provides a ranked order list of possible root causes for monitoring teams to investigate. MonitorRank uses the historical and current time-series metrics of each sensor as its input, along with the call graph generated between sensors to build an unsupervised model for ranking. Experiments on real production outage data from LinkedIn, one of the largest online social networks, shows a 26% to 51% improvement in mean average precision in finding root causes compared to baseline and current state-of-the-art methods.

Categories and Subject Descriptors: C.4 [Performance of Systems]: Modeling Techniques; I.2.6 [Artificial Intelligence]: Learning; D.2.8 [Software Engineering]: Metrics

Keywords: call graph, monitoring, service-oriented architecture, anomaly correlation

1. INTRODUCTION

The modern web architecture consists of a collection of *services*, which are a set of software components spread across multiple machines that respond to requests and map to a specific task [25]. That is, a user request is load balanced to a front end service, which fans out requests to other services to collect and process the data to finally respond to the incoming request. The callee services of this request can also call other services, creating a call graph of requests.

For example, LinkedIn, one of the largest online social networks, has a recommendation feature called "People You May Know" that attempts to find other members to connect with on the site [20]. To show this module, several services are called: a web server wrapped as a service to receive and parse the member's request, a recommendation service that receives the member id from the web server and retrieves recommendations, and finally a profile service to collect metadata about the recommended members for decorating the web page shown to users.

In this web architecture, a service is the atomic unit of functionality. Such an architecture allows easy abstraction and modularity for implementation and reuse, as well as independent scaling of components. Modern websites consist of dozens and often hundreds of services to encompass a breadth of functionality. For example, to serve its site, LinkedIn runs over 400 services on thousands of machines in multiple data centers around the world.

Naturally, for business reasons, it is important to keep the services running continuously and reliably, and to quickly diagnose and fix anomalies. As site availability is of paramount concern due to its revenue-impacting nature, web properties have dedicated monitoring teams to inspect the overall health of these services and immediately respond to any issue [8]. To that end, these services are heavily instrumented and alert thresholds are set and aggressively monitored.

However, if an alert is triggered or an anomalous behavior is detected, it is difficult and time-consuming to find the actual root cause. First, considerable user functionality and the various types of services make the request dependencies between services complex. That is, these dependencies follow the transitive closure of the call chain, where services reside on multiple machines, can be stateless or stateful, and could call other services serially or in parallel. Due to this complexity in dependencies, a monitoring team has to maintain deep domain knowledge of almost every service and its semantics. However, it is very hard for the team to keep updating such knowledge, particularly when the site evolves quickly through rapid deployment of new features. Second, the total number of services is large and each service can generate hundreds of metrics. This means that mining and understanding the metrics for diagnosis is time-consuming. Last, even though heuristics and past knowledge are applied by the monitoring teams to narrow the search space in the event of an anomaly, evaluation can only be done in human time—which is, at a minimum, in the tens of seconds per possible root cause. For these reasons, automated tools to help narrow down the root cause in the event of an anomaly can substantially reduce the time to recovery.

This paper introduces *MonitorRank*, a novel unsupervised algorithm that predicts root causes of anomalies in a service-oriented

*Work was performed while the author was interning at LinkedIn.

SIGMETRICS'13, June 17–21, 2013, Pittsburgh, PA, USA.

architecture. It combines historical and latest service metric data to rank, in real time, potential root causes. This ranked list decreases the overall search space for the monitoring team. MonitorRank is unsupervised, eliding the need for time-consuming and cumbersome training required for learning the system.

As part of our evaluation, we analyzed anomalies such as latency increases, throughput drops, and error increases in services over the course of 3 months at LinkedIn. MonitorRank consistently outperformed the basic heuristics that were employed by LinkedIn's monitoring team and current state-of-the-art anomaly correlation algorithms [23]. In terms of mean average precision, which quantifies the goodness of root cause ranking, MonitorRank yields 26% to 51% more predictive power than any other technique we tried.

The rest of the paper is organized as follows. Section 2 provides background on service-oriented architectures and Section 3 showcases related work. In Section 4, we discuss the challenges of detecting root causes in a service-oriented architecture. We then present our approach, MonitorRank, in Section 5. Section 6 evaluates our algorithm against the current state-of-the-art techniques using labeled data from previous anomalies at LinkedIn. We close with future work in Section 7.

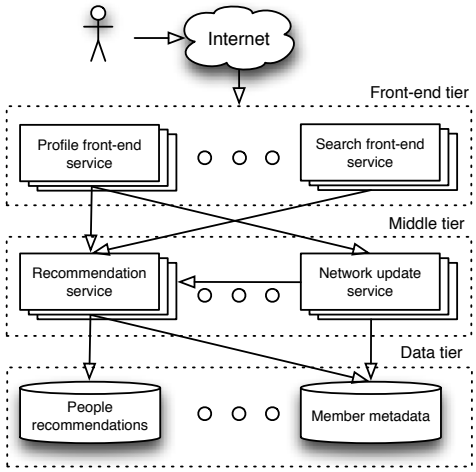


Figure 1: A generic web site architecture consisting of multiple tiers of services

2. BACKGROUND

At a very high level, a website architecture consists of 3 tiers of services: the frontend, middle tier and data tier. Services act as the building blocks of the site and expose various APIs (Application programming interface) for communication. Figure 1 presents the high-level architecture of LinkedIn.

The *frontend tier*, also called the *presentation tier*, consists of services responsible for receiving requests from members and translating the requests to calls to other downstream services. In practice, most user-facing features map to only one of these frontend services. The *middle tier*, or the *application tier*, is responsible for processing the requests, making further calls to the data tier, then joining and decorating the results. The services at this tier may also communicate among themselves. For example, a service handling the social network updates for users injects suggestions from the service in charge of recommendations. Based on the APIs called, the recommendation service may provide member, jobs, or group recommendations. Finally, the *data tier* has services that maintain a thin decoration wrapper around the underlying data store, while also performing other storage strategies such as caching and replication. Other nomenclature commonly used is *backend tier*, which encompasses both the middle and data tier.

LinkedIn runs approximately 400 services, deployed on thousands of machines in multiple data centers. These services are continuously developed and deployed by many engineers. As the number of features on the site grows, the number of services also increases. Furthermore, because each service maintains various APIs, the overall number of *sensors*, meaning <service, API> tuples, is very high.

Hence, the problem of detecting the root cause of an anomaly among many sensors demands considerable human effort. The root cause finding problem requires manual scanning of sensor logs and following all possible downstream trajectories of sensors. Domain knowledge might be used for making the diagnosis faster by restricting the candidate trajectories to a small subset of sensors. However, it is difficult to keep up-to-date with information about all of these sensors if the site is rapidly evolving. In the worst case, all teams involved in developing the downstream sensors need to be involved in finding the root cause, which hurts overall productivity.

3. RELATED WORK

Finding the root causes of anomalies has been extensively studied in various areas including chemical engineering [37, 38] and computer networks [27]. In computer networks, effort has been made on handling both real-time metrics/events and link information [4, 6, 16, 36]. However, the links in the computer network structure represent a reliable dependency. For instance, if a router has an anomaly, then all traffic flowing through it is affected. On the other hand, in the context of web sites, a sensor generating a lot of exceptions may not necessarily propagate this behavior to its callers. Due to this lossy nature in our use case, strong assumptions with respect to links do not hold.

In terms of large-scale system troubleshooting, many attempts have been made to develop middleware that efficiently finds the root causes of anomalies. In particular, VScope [32] and Monalytics [31] allow monitoring teams to consider the relationships between sensors for root cause finding. However, such middleware rely on manual rule-based configuration thereby requiring deep domain knowledge. In contrast, our algorithm aims to reduce human effort in troubleshooting without domain knowledge.

There has also been a great deal of focus on using machine learning techniques for finding root causes in systems. One methodology is to learn from historical data and find anomalies in the current data [1, 7, 18]. These methods employ supervised algorithms, in contrast to the unsupervised method we adopted. Another direction of work has focused on anomaly correlation with graph structure features [2, 11, 15, 23]. These algorithms make an assumption that a link between two nodes represents a strong dependency, which is not true in practice for sensor call graphs. Other techniques attempt to capture dependencies between nodes by using correlation [13]. This assumption too does not hold as there have been cases in various production logs at LinkedIn where correlation in normal state is not the same as in an anomalous state. MonitorRank finds dependencies between sensors based on pseudo-anomalies and uses the call graph in a randomized way.

Anomaly detection algorithms is also related to this work, as used by a clustering algorithm inside MonitorRank. Our proposed algorithm implements a heuristic method, but can easily be extended to more sophisticated algorithms, including subspace methods [17, 22, 35], matrix factorization [34], or streaming methods [9, 21, 28].

4. ROOT CAUSE FINDING PROBLEM

This section formally describes in detail the problem of finding the root cause sensor and its challenges.

4.1 Problem definition

Before defining the problem, we summarize the data available for diagnosis. Individual sensors emit metrics data (for example, latency, error count, and throughput) with a unique universal identifier. This identifier is generated in the first frontend service receiving the user request and then propagated down to the downstream callee services. This metrics data is collected and stored in a consolidated location so as to create the complete *call graph* by joining on the universal identifier. A call graph is a directed graph where each node represents a sensor, and an edge from node v_i to node v_j indicates that sensor v_i called sensor v_j. For simplicity, this paper assumes a single unweighted edge between two nodes even though there can be multiple calls between two sensors during a request. Also, though this paper presents results at a sensor level, that is, service and API combination, it can be extended to work at a coarser (for example, just a service) or finer (for example, a combination of <service, API, server> tuple) granularity.

Based on the described data, the formal description of the root cause finding problem is as follows. Suppose that an anomaly is observed in metric m of a frontend sensor v_{fe} at time t. Then, given the metric m of other sensors and the call graph, our goal is to identify sensors that caused the anomaly. To help with diagnosis, MonitorRank provides an ordered list of sensors to examine. In the best case, the first sensor presented in the ranked output of MonitorRank is the exact root cause of the anomaly after investigation. In other words, our objective is to rank sensors directly relevant to the root cause higher compared to those unrelated to it.

We restrict our focus to diagnosis after an anomaly has been reported, thereby distinguishing our work from the anomaly detection literature [3, 5, 26]. The detection of the anomaly happens through either a user report or a threshold-based alert. In both cases, MonitorRank is provided with a metric corresponding to the anomalous sensor and the approximate time range of the anomalous behavior.

4.2 Challenges

MonitorRank uses two primary pieces of data: the metric data of each sensor and the call graph between sensors. However, using just the bare-bones call graph is more challenging compared to the metric data. First, the call graph might not represent the true dependency between sensors in production. It does not account for various external factors that can influence the behavior of a service. For example, malicious bot behavior on the site can increase latency or a human error during maintenance can decrease throughput. Also, if co-located on the same hardware, two sensors may face the same problem, such as high CPU utilization, even though these sensors do not have a direct call edge between them. These examples demonstrate that similar anomalous behaviors between sensors may be independent of their relationship in the call graph.

Second, the calls between sensors are not homogeneous. A sensor can call multiple sensors in series or in parallel. Serial and parallel calls would result in different dependency type even though the schematic of the sensor calls is the same. For example, call latency for a service would be greater than the sum of its downstream calling service's latency for serial calls, but be maximum in case of parallel calls. Even the same pair of sensors can have different dependencies among the sensors. For example, the same operation of displaying recent network updates on a social network web site can generate a different call graph dependency for a user with 1000 connections than a user with just 10. The user with 1000 connections may have a bottleneck on the update generation sensor, while for a user with just 10 connections all updates might be cached and the bottleneck would instead be the profile information fetching sensor.

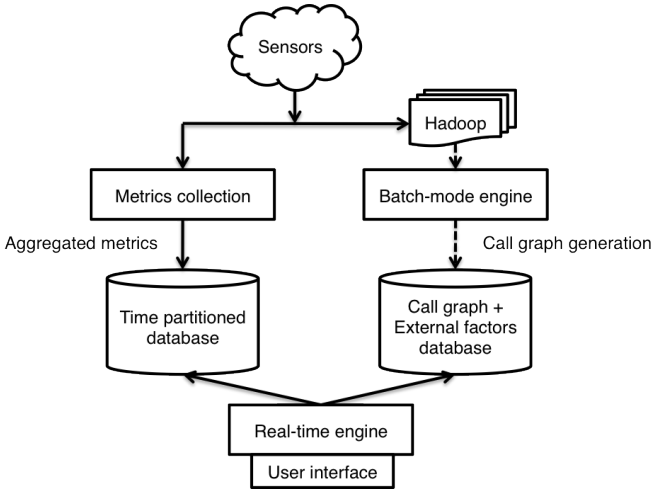


Figure 2: Sub-components required by MonitorRank. The dashed lines signify only periodic updates

Third, the dependencies between sensors are dynamic with relationships changing over time. For example, if a new algorithm is deployed to improve the running time of some API call, then the dependency between sensors, with respect to latency metric, would be different before and after a new deployment.

Due to these reasons, the call graph may not reliably represent dependencies between sensors. Therefore, the final algorithm of finding the root cause has to consider each edge of the call graph as one of the possible routes of anomaly propagation.

Our algorithm attempts to find the root cause of an anomaly as an unsupervised problem because in many cases labeled data for training may not be available. Even if the labeled data exists, we cannot guarantee the correctness of the data at any given time due to the rapidly changing underlying system.

Finally, the algorithm should display the results quickly. Because the user is provided a degraded experience during the period of diagnosis, the recovery time is of utmost importance. Hence, the algorithm is required to have subquadratic runtime, with any call graph computation being faster than $O(N^2)$, where N, the number of sensors, is typically large.

5. OUR APPROACH

To rank the sensors that are potentially contributing to the given anomaly, MonitorRank assigns a *root cause score* for each sensor. To give the score, our framework, illustrated in Figure 2, uses three sub-components: metrics collection system, a batch-mode engine, and a real-time engine. We briefly introduce each sub-component here and later explain them in detail in the following subsections.

First, the metrics collection system receives and aggregates metrics from all the sensors, and finally stores them into a time-partitioned database.

The batch-mode engine executes periodically on a snapshot of metric data to generate the call graph and extract the external factors. The output of this is stored back into another database. The external factors are extracted by a *pseudo-anomaly clustering* algorithm, described further in Section 5.2. Generation of the call graph periodically makes MonitorRank robust to rapid service evolution.

Last, when an anomaly occurs, the monitoring team interacts with the real-time engine via a user interface. The inputs provided are the

frontend sensor, approximate time period of anomaly, and the metric under consideration. The first sensor causing an anomaly is easy to find in the context of web sites because of their one-to-one mapping to a user-facing feature. For example, an error reading a LinkedIn message would point to the `getMessageForMember()` API call in the *inbox frontend* service. The real-time engine then loads necessary data from the two databases and provides the ranked list of root-cause sensors. With this data, the real-time engine performs a random walk algorithm, as described in Section 5.3.

5.1 Metrics Collection

As the number of sensors grows, it becomes important to standardize on the metrics naming and collection so as to help the monitoring team analyze and remediate user-facing issues quickly. All APIs in LinkedIn implement a generic interface that provide standard metrics, such as latency, throughput, and error count. Introduction of new sensors results in automatic generation of new metric data. The metrics from these sensors are passed to an agent: a simple program located on all machines, which periodically pushes the metrics to a centralized broker via a publish/subscribe system called Kafka [19]. The data in Kafka is then continuously consumed by our metrics collection application, buffered and aggregated to a coarser time granularity and then stored into a time-partitioned database. This centralized metrics database serves various applications such as a real-time monitoring visualization dashboard; a simple threshold alerting system; or the focus of this paper, that is, a root cause finding dashboard.

5.2 Batch-mode Engine

The metrics data from Kafka is also consumed by Hadoop, a batch processing system, and stored on Hadoop's distributed file system (HDFS). A regularly scheduled Hadoop job takes as its input a snapshot of the metrics data and outputs the call graph and external factors.

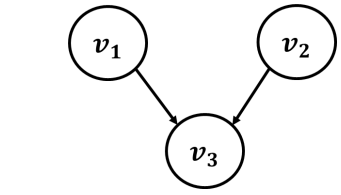
Call Graph Generation

Every individual metric data point stored on Hadoop contains the unique universal identifier introduced by a frontend sensor and then passed down along the call chain. The metric data point also contains the caller and callee sensor name. A Hadoop job joins the snapshot metrics data on the universal id and then combines the individual sensor names to generate the call graph.

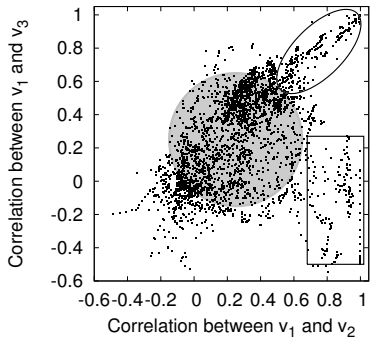
Pseudo-anomaly Clustering

However, the call graph does not reliably represent dependencies between sensors due to external factors as described in Section 4. When a certain external factor causes an anomaly, sensors affected by the external factor show highly correlated metric pattern in the anomaly time window.

For instance, if two sensors do not call the same API but are co-located, one sensor would be affected whenever the other shows anomalous behavior due to hardware problems. Moreover, as the location of the sensors would not rapidly change over time, this colocation effect can be captured from the historical metric data. Figure 3(a) shows an instance of the call graph from LinkedIn where sensors v_1 and v_2 were co-located. Figure 3(b) shows the correlation between v_1 and v_2 versus the correlation between v_1 and v_3, plotted for the error count metric. If v_1 and v_2 were correlated only through a common neighbor sensor v_3, without any external factors, most data points would have clustered into the ellipsoid area only. However, we also notice a different cluster, highlighted as a rectangle in Figure 3(b), where the correlation between v_1 and v_2 is highly independent of that between v_1 and v_3. This rectangular area



(a)



(b)

Figure 3: Effect of external factor. Even though a sensor v_3 is called by both v_1 and v_2 in (a), the correlation between v_1 and v_2 can be high even when the correlation between v_1 and v_3 is low in the rectangle area of (b). The rectangle area is caused by the colocation of v_1 and v_2.

is caused by the external factor, in this case due to their colocation. Using the above ideas, we propose a *pseudo-anomaly clustering* algorithm that groups sensors together based on historical metrics correlation, thereby capturing cases like the rectangle area.

Since the prerequisite to clustering is to detect anomalies, we use one of the various detection algorithms in literature on the historical metric data. The output of the detection algorithm is an anomalous frontend sensor, the corresponding metric and time range (moment). Because these detected anomalies may not necessarily capture only true reported anomalies, we refer to them as *pseudo-anomalies*. Then for every pseudo-anomaly we need to compute the similarity of the corresponding metric data against those of all the other sensors.

The measurement of relevance between the frontend sensor and the others, is defined by the *pattern similarity* between their corresponding metric values. If some external factor caused an anomaly in the same set of sensors, including the given seed frontend sensor, the metric patterns of those sensors should be similar with respect to the pseudo-anomaly event. Conversely, if we investigate the metric pattern similarity among sensors at pseudo-anomaly moments and find a group of sensors that usually show high pattern similarity in common, then we can assume the influence of some external factor.

Conditional clustering. To consider external factors in our algorithm for each frontend sensor, we need to find a group of sensors that are commonly related during our detected pseudo-anomalies. This problem is slightly different from a conventional clustering problem for the following reasons. First, the algorithm needs to solve a separate clustering problem for each frontend sensor because the detected pseudo-anomaly time ranges vary depending on the seed sensor. Second, only groups that consist of sensors with high pattern similarity need to be considered. Since we can discard time ranges where the metric values for the seed sensor do not look similar to those of the other sensors, the false positive rate of the underlying detection algorithm becomes less critical. Hence, the

clustering problem is conditioned on pseudo-anomaly moments of the seed sensor, as well as the pattern similarity scores of the other sensors with respect to the seed sensor.

Based on the above requirements, the clustering problem is formulated as follows. Let pseudo-anomaly time ranges (moments) be $t_1, t_2, \cdots, t_M$ for a given frontend sensor v_{fe}. For each moment t_k, we denote the pattern similarity between the frontend sensor v_{fe} and all the other sensors by $S^{(t_k)} = [S_1^{(t_k)}, S_2^{(t_k)}, \cdots, S_{|V|}^{(t_k)}]$ such that each $|S_i^{(t_k)}| \leq 1$. Without loss of generality, we set $v_1 = v_{fe}$. The objective is then to find clusters from $S^{(t_k)}$, where the number of cluster is not specified. We aim to create a sparse set of sensors per cluster, such that the pattern similarity scores within the group is very high. Also, even though we learn the clusters from the historical data, matching of a given current metric data to a cluster should be possible in real-time. Finally, assuming that external factors consistently influence the same set of sensors for a reasonable time, we only consider clusters supported by large number of samples.

Suppose that the seed frontend sensor and other sensors $n_1, \cdots, n_c$ are affected by the same external factor. Then, at pseudo-anomaly moment t when this external factor causes an anomaly,

$$S_i^{(t)} \sim \begin{cases} \mu^{(t)} + \epsilon^{(t)} & \text{for } i = n_1, n_2, \cdots n_c \\ \mathcal{N}(0, \delta^2) & \text{otherwise} \end{cases} \quad (1)$$

for a shared variance δ^2, the average relevance $\mu^{(t)}$ with respect to each pseudo-anomaly moment t, and a small error $\epsilon^{(t)}$. We also assume that $\delta < \mu^{(t)}$ to make external factors distinguishable from random effects. In other words, if sensors are affected by the same external factor, their pattern similarity scores with regard to the seed sensor will be close and high; otherwise, sensors that do not belong to a given cluster show low pattern similarity scores.

These assumptions simplify the real-world, but make sense when we revisit Figure 3(b). We ignore the cluster in the center because it represents low similarity between each sensor and the frontend one sensor (that is, mixed with random effects). The remaining two areas (rectangle and ellipsoid) agree with our assumption since sensors unrelated to a given pseudo-anomaly represent small pattern similarity scores. Also the related sensors in these areas show high similar scores between each other. In practice, this phenomenon was also observed for other combinations of sensors, thereby validating our assumption.

Algorithmic detail. Now we describe the pseudo-anomaly clustering algorithm in detail. The inputs to the clustering algorithm are (a) a seed frontend sensor v_{fe} (that is, v_1), (b) various pseudo-anomaly moments from historical data ($t_1, \cdots, t_k$), and (c) pattern similarity scores of the sensors ($S^{(t_1)}, \cdots, S^{(t_k)}$). The historical data is limited to recent few weeks thereby restricting the amount of data to handle. The pseudo-anomaly moments where the pattern similarity scores of all services represent low values are filtered out. This way false positive moments produced by the detection algorithm can be removed.

We then capture the sensors of high pattern similarity scores with respect to each of the cleaned pseudo-anomaly moments. Using the assumption in (1), we categorize sensors (except for the frontend sensor) into two groups based on the pattern-similarity scores of the sensors. While one group consists of sensors representing low pattern similarity scores (near zero), the other group contains the sensors showing high pattern similarity scores (near $\mu^{(t)}$ in (1)). We refer to the latter group as the cluster for the given pseudo-anomaly moment. Finally, we make this cluster sparse to maintain only key sensors with regard to a certain external factor.

To fulfill the above objective, we formulate the optimization problem with respect to each pseudo-anomaly moment t_k. Let x_i be the variable indicating whether the sensor v_i belongs to a cluster of the moment t_k for $i = 2, \cdots, |V|$. That is, x_i is either 0 or 1, where $x_i = 1$ means that the sensor v_i belongs to the cluster. We do not consider x_1 because we set v_1 as a given frontend seed sensor. From (1), we then minimize not only the errors $\epsilon^{(t)}$ of sensors in the cluster but also the variance of the other sensors. Further, to make a sparse cluster, we also minimize the cardinality of x as follows:

$$\begin{aligned} \min_{\mu^{(t)}, x} \ & \frac{1}{2}||S^{(t_k)} - \mu^{(t)} x||_2^2 + \frac{\delta^2}{2} card(x) \\ \text{subject to} \ & x \in \{0,1\}^{|V|-1} \\ & \mu^{(t)} \geq \delta \end{aligned} \quad (2)$$

where $card(x)$ means the cardinality of a vector x and $\mu^{(t)}$ indicates the average relevance of sensors in the cluster.

To reformulate this problem by L1-regularization and convex relaxation, we obtain the following convex optimization problem:

$$\begin{aligned} \min_{\mu^{(t)}, x} \ & \frac{1}{2}||S^{(t_k)} - x||_2^2 + \frac{\delta^2}{2}|x|_1 \\ \text{subject to} \ & 0 \leq x_i \leq \mu^{(t)} \ \ i = 2, \cdots, |V| \\ & \mu^{(t)} \geq \delta \,. \end{aligned} \quad (3)$$

Note that the minimum of the objective function value does not increase as $\mu^{(t)}$ increases. Hence, without influence on the solution of x, the variable $\mu^{(t)}$ can be dropped from the problem. This helps in simplifying the problem and the allow us to find x by Lasso [10, 29].

Once a solution (x^*) is obtained, the set of sensors $\{v_i | x_i^* > 0\}$ are referred to as an *active set* A_k with respect to pseudo-anomaly moment t_k. The active set is generated for each moment and then is run through two filtering steps. First, if the seed frontend sensor ends up with one leaf node, then we remove the corresponding active set since the high anomaly correlation in this set can be explained by propagation from the leaf-node sensor. Second, to make the number of clusters small, we filter out active sets using a support threshold value θ. The remaining active sets are then regarded as the final pseudo-anomaly clusters and are stored into the same database storing the call graph. This data is eventually used by the real-time engine while running MonitorRank.

This pseudo-anomaly clustering algorithm need not to be performed in real-time since the amount of historical data is massive. The algorithm translates into a couple of Hadoop job that are run periodically and update all the clusters in the database at once. The implementation is also inherently parallel as we can run the above jobs for each frontend sensor concurrently.

5.3 Real-time Engine

While the metrics collection pipeline and the batch-mode engine run continuously in the background, the real-time engine is responsible for serving the queries from the monitoring team. A simple user interface allows them to enter a frontend sensor, a metric, and a time period corresponding to the observed anomaly. The result is an ordered list of root-cause sensors. The interactive nature of the user interface allows the team to change their inputs continuously while getting back diagnosis guidelines in near real-time.

Pattern Similarity

One way of finding the root cause sensor in real-time is to search for sensors showing similar "patterns" in a metric to the frontend

sensor where the anomaly is observed. For instance, suppose that many errors suddenly occurred in a data tier sensor due to a timeout. These errors propagate up the sensor-call path, resulting in a similar increasing pattern throughout. Generalizing this idea, if two services have similar abnormal patterns in a certain metric, we can imagine that the patterns may be caused by the same root cause. In particular, the metric data of two sensors can look similar when the same anomaly causes a change, regardless of whether the data looks similar in the normal situation.

Therefore, we use the similarity of the metric between the anomalous sensor and its corresponding downstream sensors as the key feature of MonitorRank. For each sensor v_i, computation of the metric pattern similarity score S_i, with respect to the anomalous sensor v_{fe}, is as follows:

$$S_i = \mathrm{Sim}\left(m_i, m_{fe}\right) \tag{4}$$

$\mathrm{Sim}(\cdot,\cdot)$ is a fixed similarity function between two time series data (for example, correlation) and $m_i \in \mathcal{R}^T$ represents the time series metric data of sensor v_i during a time period of length T, which is provided as an input. This similarity score S_i signifies the relevance of the service v_i to the given anomaly.

Random Walk Algorithm

Although the metric pattern similarity score is useful for detecting the relevance of sensors to the given anomaly, using just this score can result in false positive root causes. This is because sensors with no relation to the anomaly may still have a very high similarity score with the anomalous sensor. For example, seasonal effects can result in nearly all sensors having a similar increase in throughput, thereby resulting in a high similarity score. As correlation does not necessarily imply causality, the similar anomalous metric pattern between two sensors cannot be the only factor to determine whether one sensor is causing an anomaly in the other. Instead, these pairs are taken as candidates and further addition of the call graph is required to rank the candidate sets better.

The call graph is incorporated using a randomized algorithm, keeping in mind that it does not reliably represent the dependencies between sensors. The basic idea of our approach is to do a random walk over the call graph depending on the similarity score. More specifically, sensors are selected in sequence by randomly picking up the next sensor among the neighbors in the call graph. The pickup probability of each neighbor is proportional to its relevance to a given anomaly, which is captured by the pattern similarity score. We then assume that more visits on a certain sensor by our random walk implies that the anomaly on that sensor can best explain the anomalies of all the other sensors. Under this assumption, the probability of visiting each sensor is regarded as the root cause score of the sensor in our algorithm. This allows us to blend the relevance of each sensor, for a given anomaly, into the call graph.

Our procedure is analogous to the *Weighted* PageRank algorithm [24, 33]. However, we do not consider sensors that look uncorrelated with a given anomaly. Therefore, MonitorRank uses the *Personalized* PageRank algorithm [14], by taking teleportation probability (preference vector) as determined by the given anomaly. The nature of the this PageRank algorithm allows teleportation to any sensor uniformly at random.

Basic setup. The random walk algorithm is defined as follows. Let the call graph be $G =< V, E >$, where each node indicates a sensor and each edge $e_{ij} \in E$ is set to 1 when sensor v_i calls sensor v_j. We assume that there is no self edge, that is, $e_{ii} \notin E$.

The other inputs to the algorithm include an anomalous frontend sensor node v_{fe} and the pattern similarity score S_i of each sensor

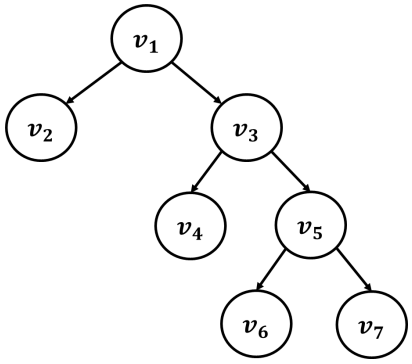


Figure 4: Natural trapping in the call graph. Due to the nature of downward sensor calls (from frontend to backend tier), when the random walker falls into the branch of v_3, the random walk cannot visit v_2 until the next teleportation even if there is no sensor related to the given anomaly in $v_4 \sim v_7$.

$v_i \in V$. Again, without loss of generality, let v_1 be v_{fe}. For convenience, the similarity score vector of all the sensors is denoted by $S = [S_1, \cdots, S_{|V|}] \in (0,1]^{|V|}$. For the random walker to visit each node v_i proportionally to its pattern similarity score S_i, the strength of each edge e_{ij} is assigned as S_j. The transition probability matrix Q of this movement is represented by:

$$Q_{ij} = \frac{A_{ij} S_j}{\sum_j A_{ij} S_j} \tag{5}$$

for $i, j = 1, 2, \cdots, |V|$ for the binary-valued adjacency matrix A of the call graph G.

Self-edges. By definition, the random walker is enforced to move to another node even if the current node shows a higher pattern similarity score and all the neighboring nodes do not. This type of movement increases the stationary probabilities of nodes unrelated to the given anomaly.

To avoid forcibly moving to another node, a self-edge is added to every node. The random walker then stays longer on the same node if there is no neighboring node of high similarity score. A strength value of the self edge is determined by the pattern similarity score of the target node and its neighboring nodes. Specifically, for each node v_i, the strength of the corresponding self edge e_{ii} is equal to the similarity score S_i subtracted by the maximum similarity score of child nodes ($\max_{j:e_{ij}\in E} S_j$). This way the random walker is encouraged to move into the nodes of high similarity scores, but is prevented from falling into the nodes irrelevant to the given anomaly. As an exception we do not add a self-edge to the frontend node, that is, $e_{11} = 0$, as the random walker does not need to stay at v_1.

Backward-edges. When the random walker falls into nodes that look less relevant to the given anomaly, there is no way to escape out until the next teleportation. As the direction of an edge in the call graph tends to be from the frontend to the backend, the random walker is likely to be naturally trapped inside branches of the call graph. In Figure 4, suppose that an anomaly is observed at the frontend sensor v_1 and node v_2 is the only sensor relevant to the given anomaly. The resulting pattern similarity score of node v_2 with regard to the given anomaly is even higher than those for the other nodes $v_3 \sim v_7$. If the pattern similarity scores of nodes $v_3 \sim v_7$ are not negligible, the random walker can fall into the right part of descendants in Figure 4 (nodes $v_3 \sim v_7$). In this case, the random walker would stay on the nodes $v_3 \sim v_7$ until the next random teleportation occurs, no matter how irrelevant the nodes are with respect to the given anomaly.

To resolve this issue, backward edges are added so that the random walker flexibly explores nodes of high pattern similarity score. While random teleportation in the Personalized PageRank algorithm makes a random walker explore globally, the backward edges allow the random walker to explore locally. By adding these backward edges, we achieve the restriction imposed by the call graph, but with the added flexibility to explore. However, because we add backward edges by means of local teleportation, less strength is set on each backward edge than the strength of true edges ending up to the same node. For every pair of nodes v_i and v_j, such that $e_{ij} \in E$ and $e_{ji} \notin E$, while the strength of e_{ij} is equal to S_j, the strength of e_{ji} is set as ρS_i for some constant $\rho \in [0, 1)$. If the value of ρ is high, the random walker is more restricted to the paths of the call graph, that is, from upstream to downstream. Alternately, when the value of ρ is low, the random walker explores the nodes with more flexibility. If the call graph represents a true dependency graph between sensors, we would set ρ higher, and vice versa.

To incorporate both backward and self edges, we define a new real value adjacency matrix A' with the call graph G and the similarity score S as follows:

$$A'_{ij} = \begin{cases} S_j & \text{if } e_{ij} \in E \\ \rho S_i & \text{if } e_{ji} \in E, e_{ij} \notin E \\ \max(0, S_i - \max_{k:e_{jk} \in E}) & j = i > 1 \end{cases} \tag{6}$$

Using A', the new transition probability matrix P is defined as:

$$P_{ij} = \frac{A'_{ij}}{\sum_j A'_{ij}} \tag{7}$$

for $i, j = 1, 2, \cdots, |V|$.

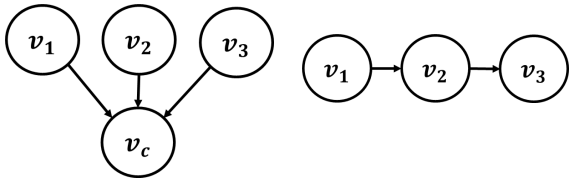
Given an anomaly and related frontend sensor v_1, the Personalized PageRank vector (PPV) $\pi_{PPV} \in (0, 1]^{1 \times |V|}$ serves as the root cause score for each node. For a preference vector u of the random walk (personalized teleportation probability vector), the pattern-similarity score S is used for $v_2, \cdots, v_{|V|}$. That is, $u_i = S_i$ for $i = 2, 3, \cdots, |V|$. In this way, a random walker jumps more to anomaly-related sensors whenever the random teleportation occurs. Also because staying at the frontend sensor v_1 is out of focus for the random walk, the value in the preference vector corresponding to the frontend sensor v_1 is assigned as zero ($u_1 = 0$). Even though we do not make a random teleportation to the v_1, we still require this frontend sensor v_1 as a connecting node between the neighboring nodes of the v_1.

Once we determine the transition probability P and the preference vector u, PPV is obtained as follows:

$$\pi_{PPV} = \alpha \pi_{PPV} P + (1 - \alpha) u\,. \tag{8}$$

where α is a random teleportation probability in $[0, 1]$. Similarly for the backward edge probability constant ρ, we set the α higher when the call graph is thought of as a true dependency graph. In the extreme case that $\alpha = 0$, this random walk model is equivalent to using only the metric pattern similarity.

Why the random walk algorithm?. MonitorRank is inspired by the idea that biological organisms efficiently search for their target (for example, the source of scent) by performing a random walk, even though the sensory data from the organisms is not reliable [12, 30]. Furthermore, a random walk algorithm is also analogous to human behavior during diagnosis. When an engineer from the monitoring team has no knowledge about the system, except for the call graph, one of natural diagnosis methods is to randomly traverse sensors by following the call graph with preferentially looking at misbehaving nodes. Hence, the stationary probability by random walk represents the behavior of multiple non-expert engineers monitoring each sensor node. A high stationary probability for a certain node implies that this node is "democratically" the most important to investigate without prior knowledge.



(a) Common child example (b) Chain example

Figure 5: Graph examples that benefit from the random walk model

The PageRank algorithm tends to rank the nodes with more edges higher, as such nodes have more incoming paths. However, our approach naturally mitigates the effect of this advantage to some extend. Because we allow a self-edge for every node except for a given frontend node, a node having low connections, but showing similar anomaly patterns to the frontend sensor, can also high stationary probability by acting like a dead end.

The random walk algorithm can provide a higher score to a common child sensor node of some frontend sensors (that is, a common backend sensor called by the frontend sensors) if all of the frontend sensors have the same anomalies. For example, think of sensor v_c called by three sensors v_1, v_2, and v_3 in Figure 5(a). Then suppose that v_c causes an anomaly to propagate to all the other sensors $v_1 \sim v_3$ and the anomaly is observed on the sensor v_1. Because all the sensors show anomalies caused by the same root cause, the pattern similarity scores of the sensors with respect to the anomaly-observed sensor v_1 are similarly high, that is, $S_2 \approx S_3 \approx S_c$. The random walk algorithm lifts up the rank of the common sensor v_c, the actual root cause of the given anomaly.

Also, if the similarity function is correct in extracting only sensors relevant to a given anomaly and there exists an explicit anomaly causal path chain as visualized in Figure 5(b), we can prove that our random walk algorithm can successfully find the exact root cause.

THEOREM 1. *Suppose that a set of nodes $V' \subset V$ consists of a chain graph such that $v'_1 \to v'_2 \cdots \to v'_m$ for $m = |V'| \geq 1$. If $S_{v'} = p \leq 1$ for $v' \in V'$ and $S_v = 0$ for $v \notin V'$, then $\pi_{v'_m} > \pi_{v'_i}$ for $i = 1, 2, \cdots, m - 1$.*

PROOF. See Appendix. □

Finally, by overlaying multiple transition probability matrices and preference vectors the random walk approach can handle the situation where-in anomalies are seen on multiple frontends simultaneously. We do not implement and test this scenario in this paper due to its rare occurrence in production.

External Factors

The random walk algorithm that we described so far incorporates the call graph generated by the batch-mode engine with the current metric data. On the other hand, the batch-mode engine also learns external factors not captured by the call graph. MonitorRank blends the pseudo-anomaly clusters with the random walk algorithm by finding the best-match cluster with the current metric data and giving more scores to sensors in the selected cluster.

The selection of cluster, for a given anomaly and frontend sensor, is based on Gaussian assumption, as described in (1). That is, if the current similarity score is $S = \{S_1, \cdots, S_{|V|}\}$ and the root cause of the current anomaly is the common external factor of some pseudo-anomaly cluster C, then the pattern similarity scores of sensors in C

would be higher than any sensor not in C. MonitorRank selects the best-fit cluster C^* by comparing the minimum score among sensors in each cluster to the maximum score of the others as follows:

$$C^* = \arg\max_C \frac{\min_{c \in C} S_c}{\max_{c' \notin C} S_{c'}} . \quad (9)$$

Once the best-fit cluster C^* is selected, it is combined with the random walk algorithm. If the pattern similarity score ratio described in (9) is less than 1 or the average of pattern similarity scores of sensors in C^* is less than δ, we discard the selected cluster C^* and run the random walk algorithm as if C^* were not chosen. Otherwise, by regarding the average pattern similarity score of sensors in C^* as the pattern similarity score of the external factor corresponding to C^*, we add this average score to S_c for every sensor $c \in C^*$. In this way, we leverage the fact that engineers in the monitoring team examine the sensors related to the external factor first. After adding the average score, we run the same random walk algorithm to finally obtain the rank of sensors to examine.

Because the procedure described in (9) scans S_i once for each cluster C, it requires $O(N_C|V|)$ time for the number of clusters N_C. As $N_C \ll |V|$ in general, it takes time even less than $O(|V|^2)$. Furthermore, the random walk algorithm takes $O(|E|)$ time, which is also much faster than $O(|V|^2)$ (particularly in a sparse call graph). Therefore, overall, MonitorRank requires only $O(N_C|V| + |E|)$ time, which is feasible in real time.

6. EXPERIMENTS

In this section, we evaluate MonitorRank by using real-world metrics and anomalies at LinkedIn. Moreover, we decompose our algorithm into three subcomponents – pattern similarity, pseudo-anomaly clustering, and random walk on the call graph, and investigate the effect of each subcomponent.

6.1 Experimental Setup

Datasets and Implementation

In LinkedIn, there is a dedicated team monitoring site behavior and detecting any anomalies. When an anomaly occurs, the monitoring team first tries to resolve it by manually going through the list of sensors, and in the worst case, fall back to alerting the engineering team responsible for the anomalous sensor. Over time this team has stored these anomalies and corresponding list of ranked root causes in a central site-issue management system. This dataset is used for evaluation of MonitorRank. The following three primary metrics were used for our evaluation:

- Latency: Average of latency of the sensor call over a minute (25 examples)
- Error-count: The number of exceptions returned by the sensor call in a minute (71 examples)
- Throughput: The number of sensor requests in a minute (35 examples)

The metric data is aggregated up to a fixed coarser time granularity (1 minute) thereby providing predictable runtime for our analysis. This granularity also helps in reliably catching the change of metrics, making it resilient to issues such as out-of-sync clock times.

Note that the directions of anomaly propagation in the call graph are different depending on the metrics. An anomaly in throughput may propagate from the frontend sensor to the backend sensor, whereas an anomaly in error count or latency is likely to propagate in the opposite direction, from the backend to the frontend. Therefore, when applying our algorithm on the test sets, we use the original directed call graph (frontend $\rightarrow$ backend) for throughput, while we use the reverse direction of call graph (backend $\rightarrow$ frontend) for latency and error-count.

For our pattern similarity function, we use the sliding-window correlation. In particular, we compute the correlation of two given metrics on a time window of fixed length. By moving this window over the anomaly time range, we average the correlation values computed by each time window slot. For our experiments, we used 60 minute time window after manually trying out various candidates between $10 \sim 120$ minutes. The 60 minute time window is long enough to distinguish a sudden change in the metric from the normal variance of the metric. But the 60 minute period is also short enough to highlight the local change caused by anomalies rather than compare the long-term trend in the metric of normal state.

MonitorRank requires both historical and current metrics of sensors. For experiments, we use the time period of anomaly marked by the monitoring team. However, we found the time period of an anomaly ranging between 10 minutes and 3 hours (for example, a data center outage) in our experimental data. If the specified anomaly time period is less than the 60 minutes, we extend the anomaly time period to 60 minutes by inserting the data before the time period as much as required. For historical data, we group each metric week by week and use the data two weeks prior to the given anomaly.

Finally, with regard to an anomaly detection algorithm (for pseudo-anomalies), we apply a heuristic method that determines anomalies based on the ratio of current metric value and the maximum value in the previous time window, that is, previous 60 minutes in our case. For instance, if we set a threshold as 1.2, when the maximum value of a give metric in previous 60 minutes is 100, we label the current time period as an anomaly if the current metric value is over 120. This way can capture abrupt increases in metric values. Similarly, we can define the anomalies that show sudden decrease in metric values.

Note that we may not require the best quality of anomaly detection algorithms. Most of the detection algorithms aim to reduce the false positives as much as possible, but we are tolerant of these false positives because our objective is to cluster sensors. Furthermore, as our clustering algorithm relies on the metric pattern similarity between frontend and backend sensors, the false positive anomaly moments which are not correlated with other sensors would be naturally filtered out.

Baseline Methods

For the purpose of comparison, we introduce some baseline methods, which includes three heuristic algorithms and one algorithm using the call graph as a dependency graph. We describe each algorithm as follows:

- *Random Selection* (RS): A human without any domain knowledge will examine sensors in random order. We mimic this behavior by issuing random permutations.
- *Node Error Propensity* (NEP): Under the assumption that an anomaly on a certain sensor would produce errors on the same sensor, one can view the error count on a time window as a measure of anomaly.
- *Sudden Change* (SC): A natural way for a human to find root causes is to compare the metrics in the current and previous time windows and check any sudden change between the two time windows. For example, if there is a sudden latency increase at a certain time, then the average latency after this time should be notably higher than before. We therefore define the ratio of average metrics on both the time periods and refer to this ratio as the root cause score of each sensor.

	RS	NEP	SC	TBAC	MonitorRank	Improvement compared to (RS, NEP, SC)	Improvement compared to TBAC
Latency							
PR@1	0.120	0.160	0.560	0.840	**1.000**	78.6%	19.0%
PR@3	0.120	0.093	0.613	0.787	**0.920**	50.1%	16.9%
PR@5	0.144	0.112	0.650	0.756	**0.852**	31.1%	12.7%
MAP	0.143	0.206	0.559	0.497	**0.754**	34.9%	51.8%
Error Count							
PR@1	0.042	0.133	0.577	0.746	**0.901**	56.2%	20.8%
PR@3	0.049	0.493	0.653	0.552	**0.850**	30.2%	54.0%
PR@5	0.084	0.616	0.743	0.482	**0.813**	9.4%	68.7%
MAP	0.091	0.465	0.659	0.442	**0.818**	24.1%	85.0%
Throughput							
PR@1	0.229	0.000	0.086	0.971	**1.000**	336%	3.0%
PR@3	0.200	0.667	0.133	0.962	**0.990**	34.9%	2.9%
PR@5	0.183	0.800	0.149	0.897	**0.971**	21.4%	8.2%
MAP	0.210	0.634	0.210	0.714	**0.779**	22.9%	9.1%

Table 1: Performance of each algorithm on each test set: MonitorRank outperforms the baseline methods in every case.

- *Timing Behavior Anomaly Correlation* (TBAC): For a non-heuristic baseline method, we use the method of anomaly correlation using a dependency graph [23]. Because this algorithm works with metric correlations between sensors and the call graph, it is comparable to our experimental setting. The difference compared to our approach is that this algorithm regards the call graph as a reliable directed acyclic dependency graph (DAG).

Evaluation Metric

To compare MonitorRank to the baseline methods, we require appropriate evaluation metrics. All the algorithms provide a rank of sensors with respect to each anomaly case. We refer to the rank of each sensor v_i with respect to an anomaly a as $r_a(i)$ and define the indicator variable $R_a(i)$ to represent whether sensor i is the root cause of an anomaly a or not (that is, either 0 or 1). To quantify the performance of each algorithm on a set of anomalies A, we use the following metrics:

- *Precision at top* K (PR@K) indicates the probability that top K sensors given by each algorithm actually are the root causes of each anomaly case. It is important that the algorithm captures the final root cause at a small value of K, thereby resulting in lesser number of sensors to investigate. Here we use $K = 1, 3, 5$. More formally, it is defined as

$$\text{PR@}K = \frac{1}{|A|} \sum_{a \in A} \frac{\sum_{i:r_a(i) \leq K} R_a(i)}{\min\left(K, \sum_i R_a\left(i\right)\right)}. \qquad (10)$$

- *Mean Average Precision* (MAP) quantifies the goodness of a give rank result by putting more weight on the higher rank result (where high rank sensor, which has lower $r_a(i)$, means that this sensor is more likely to be a root cause). It can be formulated as follows:

$$\text{MAP} = \frac{1}{|A|} \sum_{a \in A} \sum_{1 \leq r \leq N} \text{PR@}r \qquad (11)$$

  for the number of sensors N.

Note that PR@K is a measurement on each anomaly case. Hence, we compute the average of PR@K per test set (latency, error count, and throughput) to represent the overall performance of an algorithm. On the other hand, MAP quantifies the overall performance of an algorithm per test set by itself.

6.2 Performance Evaluation

First, we evaluate MonitorRank and all the baseline methods on each test set, corresponding to latency, error count, and throughput metrics. For every anomaly case in the test sets, each algorithm gives the root cause rank of sensors. We can evaluate the given rank in terms of the evaluation metrics PR@1, PR@3, PR@5, and MAP.

Table 1 compares the performance of algorithms for each test set. In every test set and evaluation metric, MonitorRank outperforms the baseline methods by a large factor. When we consider the average MAP metric on all the test sets, the improvement over the non-heuristic baseline method (TBAC) is approximately 51.4%. If we compare to the best heuristic method (RS, NEP, and SC) on each test set, MonitorRank increases performance by 25.7% on average. Also in terms of PR@K, MonitorRank consistently represents better predictive power compared to the other methods. More specifically, MonitorRank improves 49.7% and 24.5% prediction accuracy (PR@K) on average, in comparison to the heuristic method and TBAC, respectively.

However, while TBAC method shows an unbalanced performance depending on rank (better performance at top-K than overall), MonitorRank shows the consistent performance regardless of the rank. Moreover, even at the top-K root cause results, MonitorRank outperforms TBAC, although both methods incorporate the call graph based on the metric correlation.

Finally, in terms of PR@K, the performance decreases in some cases as K increases (for example, in the case of MonitorRank in every test set). This drop might seem counter-intuitive because if we cover more services by increasing K then the probability that the root cause sensor is chosen among the K should increase as well. However, in some cases of our test set, multiple sensors are labeled as root causes for a given anomaly. When such multiple root cause sensors exist, the decreasing trend of PR@K makes sense as more selection can cause more false-positives.

6.3 Analysis of Our Approach

In this section we provide a detailed analysis of the subcomponents of the MonitorRank algorithm. Recall that our algorithm consists of three main parts: pattern similarity, a random walk on the call graph, and pseudo-anomaly clustering. We perform the same experiment as the previous section, but use the following algorithms:

- Pattern-Similarity-Only (PS) uses only the pattern similarity score without the call graph or the pseudo-anomaly clustering
- With-Random-Walk (PS+RW) runs the random walk algorithm on the call graph without pseudo-anomaly clustering
- With-Pseudo-Anomaly-Clustering (PS+PAC) uses the pseudo-anomaly clustering without using the random walk algorithm
- With-All (ALL) represents the full version of MonitorRank

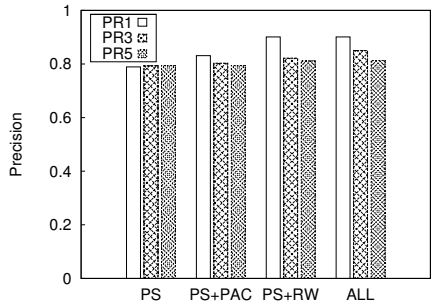


Figure 6: Effect of each subcomponent. Both the pseudo-anomaly clustering and the random walk algorithm are useful for the improvement of performance, while the effect of using the call graph is larger than the pseudo-anomaly clustering.

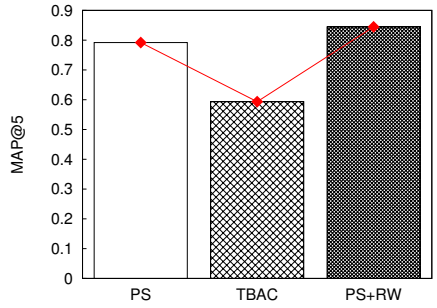


Figure 7: Effect of graph algorithm. Randomized use of the call graph improves performance, but the deterministic algorithm may degrade performance.

By running these subcomponents incrementally, we can compare the effect that each subcomponent has on the final results.

Figure 6 illustrates the effects of subcomponents by showing PR@K for $K = 1, 3, 5$ on each algorithm. For the purpose of visualization, we show the average of PR@K from all three test sets. We note that the pattern similarity plays an important role in the performance of MonitorRank, but using the call graph (random walk) lifts the performance by a significant factor. In particular, the With-Random-Walk (PS+RW) makes 14.2% improvement over the Pattern-Similarity-Only (PS) in PR@1. This improvement is achieved because the inclusion of the call graph limits candidate root cause sensors. If some sensor represents a high pattern-similarity score but its neighboring sensors do not, the random walk algorithm naturally adjusts the final root cause score to the lower value. In contrast, if sensors connected in the call graph show high pattern-similarity scores together, the root cause scores of the sensors would be properly leveraged by the random walk algorithm.

On the other hand, the pseudo-anomaly clustering seems helpful in finding the root causes of anomalies, but its effect is not as much as the random walk algorithm. When comparing the performance of With-All (ALL) and With-Random-Walk (PS+RW) in Figure 6, both algorithms (ALL and PS+RW) show similar performance in PR@1 and PR@5. However, we observe about 5% improvement (ALL over PS+RW) for PR@3 due to external factors being included. When an anomaly is caused by an external factor, using the call graph has no effect on finding all the sensors related to the external factor. In this case the With-Random-Walk (PS+RW) algorithm results in some false positives because of missing external factors. In contrast, the pseudo-anomaly clustering algorithm captures all the sensors relevant to the external factor and produce higher root cause scores. Thus, using With-All (ALL) ranks sensors relevant to the external factor are higher than the sensors that are regarded as false positives in With-Random-Walk (PS+RW).

Finally, we also investigate the contribution of the random walk algorithm on the call graph to capture non-explicit dependencies between sensors as described in Section 4. For this investigation, we compare the following algorithms: Pattern-Similarity-Only (PS), With-Random-Walk (PS+RW), and Timing-Behavior-Anomaly-Correlation (TBAC). TBAC assumes that the call graph is a strong dependency graph. Figure 7 compares the performance of each algorithm in terms of MAP@5, which is the average value of PR@$1 \sim 5$. The randomized algorithm (PS+RW) takes advantage of the call graph to find root causes better than the Pattern-Similarity-Only (PS). However, TBAC, which regards the call graph as a dependency graph, shows poorer performance, even compared to the Pattern-Similarity-Only (PS). This result is particularly interesting because an invalid assumption on the dependency between sensors can make the performance worse. In conclusion, when the call graph does not reliably represent a dependency graph, the randomized algorithm (PS+RW) on the call graph is helpful in finding root causes and we observe that treating the call graph as a strong dependency graph can be harmful in this context.

7. CONCLUSION

In this paper, we addressed the problem of finding the root causes for a given anomaly in a service-oriented web architecture. More specifically, when an anomaly is detected on a website's frontend sensor (that is, a service and API combination), our goal is to offer a ranked list of sensors to examine, with the higher ranked sensors more likely to be the root cause. This problem is challenging because the call graph, composed of API calls, does not reliably depict a dependency graph due to missing external factors.

To use call graph to increase the rank quality, we propose a random walk algorithm with preference given to similar looking sensors. Furthermore, to capture external factors not obtained by the call graph, we introduce a pseudo-anomaly clustering algorithm on historical data. We combine the two features by running the computationally intensive clustering algorithm offline and the random walk algorithm online, and provide quick results in the event of a user-facing outage. We evaluate our algorithm on production labeled anomaly dataset from LinkedIn and show significant improvement over baseline models.

We also found that even though the call graph does not represent a true dependency graph of sensors, it can be used to extract useful information during ranking. This extraction method is generic and can be leveraged by other applications, such as data center optimization.

References

[1] T. Ahmed, B. Oreshkin, and M. Coates. Machine Learning Approaches to Network Anomaly Detection. In *SysML*, 2007.

[2] A. Arefin, K. Nahrstedt, R. Rivas, J. Han, and Z. Huang. DIAMOND: Correlation-Based Anomaly Monitoring Daemon for DIME. In *ISM*, 2010.

[3] M. Basseville and I. V. Nikiforov. *Detection of Abrupt Changes - Theory and Application*. Prentice-Hall, 1993.

[4] A. T. Bouloutas, S. Calo, and A. Finkel. Alarm Correlation and Fault Identification in Communication Networks. *TCOM*, 42(2–4):523–533, 1994.

[5] V. Chandola, A. Banerjee, and V. Kumar. Anomaly Detection: A Survey. *CSUR*, 41(3):15:1–15:58, 2009.

[6] C. S. Chao, D. L. Yang, and A. C. Liu. An Automated Fault Diagnosis System Using Hierarchical Reasoning and Alarm Correlation. *JNSM*, 9(2):183–202, 2001.

[7] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer. Failure Diagnosis Using Decision Trees. In *ICAC*, 2004.

[8] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni. Automated Anomaly Detection and Performance Modeling of Enterprise Applications. *TOCS*, 27(3):6:1–6:32, 2009.

[9] P. H. dos Santos Teixeira and R. L. Milidiú. Data stream anomaly detection through principal subspace tracking. In *SAC*, 2010.

[10] B. Efron, I. Johnstone, T. Hastie, and R. Tibshirani. The Least Angle Regression Algorithm for Solving the Lasso. *Annals of Statistics*, 32(2):407–451, 2004.

[11] J. Gao, G. Jiang, H. Chen, and J. Han. Modeling Probabilistic Measurement Correlations for Problem Determination in Large-Scale Distributed Systems. In *ICDCS*, 2009.

[12] A. M. Hein and S. A. Mckinley. Sensing and Decision-making in Random Search. *PNAS*, 109(30):12070–12074, 2012.

[13] A. Jalali and S. Sanghavi. Learning the Dependence Graph of Time Series with Latent Factors. In *ICML*, 2012.

[14] G. Jeh and J. Widom. Scaling Personalized Web Search. In *WWW*, 2003.

[15] M. Jiang, M. A. Munawar, T. Reidemeister, and P. A. S. Ward. Dependency-aware Fault Diagnosis with Metric-correlation Models in Enterprise software systems. In *CNSM*, 2010.

[16] R. Jiang, H. Fei, and J. Huan. Anomaly Localization for Network Data Streams with Graph Joint Sparse PCA. In *KDD*, 2011.

[17] I. T. Jolliffe. *Principal Component Analysis*. Springer, second edition, Oct. 2002.

[18] M. Khan, H. K. Le, H. Ahmadi, T. Abdelzaher, and J. Han. DustMiner: Troubleshooting Interactive Complexity Bugs in Sensor Networks. In *Sensys*, 2008.

[19] J. Kreps, N. Narkhede, and J. Rao. Kafka: A Distributed Messaging System for Log Processing. In *NetDB*, 2011.

[20] D. Liben-Nowell and J. Kleinberg. The link prediction problem for social networks. In *CIKM*, pages 556–559, 2003.

[21] Y. Liu, L. Zhang, and Y. Guan. A Distributed Data Streaming Algorithm for Network-wide Traffic Anomaly Detection. In *SIGMETRICS*, 2009.

[22] A. Mahimkar, Z. Ge, J. Wang, J. Yates, Y. Zhang, J. Emmons, B. Huntley, and M. Stockert. Rapid Detection of Maintenance Induced Changes in Service Performance. In *CoNEXT*, 2011.

[23] N. Marwede, M. Rohr, A. V. Hoorn, and W. Hasselbring. Automatic Failure Diagnosis Support in Distributed Large-Scale Software Systems Based on Timing Behavior Anomaly Correlation. In *CSMR*, 2009.

[24] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, Stanford InfoLab, 1999.

[25] M. P. Papazoglou and W.-J. Heuvel. Service Oriented Architectures: Approaches, Technologies and Research Issues. *The VLDB Journal*, 16(3):389–415, July 2007.

[26] A. B. Sharma, L. Golubchik, and R. Govindan. Sensor Faults: Detection Methods and Prevalence in Real-World Datasets. *TOSN*, 6(3):23:1–23:39, 2010.

[27] M. Steinder and A. S. Sethi. A Survey of Fault Localization Techniques in Computer Networks. *Science of Computer Programming*, 53(2):165–194, 2004.

[28] S. C. Tan, K. M. Ting, and T. F. Liu. Fast Anomaly Detection for Streaming Data. In *IJCAI*, 2011.

[29] R. Tibshirani. Regression Shrinkage and Selection via the Lasso. *J. Royal. Stats. Soc B.*, 58(1):267–288, 1996.

[30] G. M. Viswanathan, S. V. Buldyrev, S. Havlin, M. G. E. da Luz, E. P. Raposo, and H. E. Stanley. Optimizing the Success of Random Searches. *Nature*, 401:911–914, 1999.

[31] C. Wang, K. Schwan, V. Talwar, G. Eisenhauer, L. Hu, and M. Wolf. A Flexible Architecture Integrating Monitoring and Analytics for Managing Large-Scale Data Centers. In *ICAC*, 2011.

[32] C. Wang, I. A. Rayan, G. Eisenhauer, K. Schwan, V. Talwar, M. Wolf, and C. Huneycutt. VScope: Middleware for Troubleshooting Time-Sensitive Data Center Applications. In *Middleware*, 2012.

[33] W. Xing and A. Ghorbani. Weighted PageRank Algorithm. In *CNSR*, 2004.

[34] L. Xiong, X. Chen, and J. Schneider. Direct Robust Matrix Factorization for Anomaly Detection. In *ICDM*, 2011.

[35] H. Xu, C. Caramanis, and S. Sanghavi. Robust PCA via Outlier Pursuit. In *NIPS*, 2010.

[36] H. Yan, A. Flavel, Z. Ge, A. Gerber, D. Massey, C. Papadopoulos, H. Shah, and J. Yates. Argus: End-to-end Service Anomaly Detection and Localization from an ISP's Point of View. 2012.

[37] F. Yang and D. Xiao. Progress in Root Cause and Fault Propagation Analysis of Large-Scale Industrial Processes. *Journal of Control Science and Engineering*, 2012:1–10, 2012.

[38] Z.-Q. Zhang, C.-G. Wu, B.-K. Zhang, T. Xia, and A.-F. Li. SDG Multiple Fault Diagnosis by Real-time Inverse Inference. 87(2):173–189, 2005.

APPENDIX

PROOF OF THEOREM 1. Let $m^* = \arg\max_i \pi_{v'_i}$. We will then show $m^* = m$.

We first show that $\max_i \pi_{v'_i} > \min_i \pi_{v'_i}$. Suppose that $\max_i \pi_{v'_i} = \min_i \pi_{v'_i}$, which implies that $\pi_{v'_i} = \frac{1}{m}$ for all $i = 1, 2, \cdots, m$. To see the stationary state at v'_m, the following statement should hold:

$$\pi_{v'_m} = \frac{\alpha}{1+\rho}\pi_{v'_{m-1}} + (1-\alpha)\frac{1}{m}, \ .$$

However, it is a contradiction because $\pi_{v'_m} = \pi_{v'_{m-1}} = \frac{1}{m}$ and $\rho < 1$. Hence, $\max_i \pi_{v'_i} > \min_i \pi_{v'_i}$.

Now suppose that $2 \leq m^* \leq m-1$. In other words, there exists l and r such that $v'_l \to v'_{m^*} \to v'_r$. At $v_{m'}$, we state the stationary probability as follows:

$$\pi_{v'_m} = \alpha\left(\frac{1}{1+\rho}\pi_{v'_l} + \frac{\rho}{1+\rho}\pi_{v'_r}\right) + (1-\alpha)\frac{1}{m}\ .$$

However, because $\max_i \pi_{v'_i} > \min_i \pi_{v'_i}$, $\pi_{v'_m} > \frac{1}{m}$. Also, by definition $\pi_{v'_m} \geq \pi_{v'_l}$ or $\pi_{v'_r}$. This means that the left-hand side is greater than the right-hand side in the above stationary probability equation, so it is contradiction. Therefore, m^* is not $2, \cdots, m-1$.

If $m^* = 1$, to see $\pi_{v'_1}$,

$$\pi_{v'_1} = \alpha\frac{\rho}{1+\rho}\pi_{v'_2} + (1-\alpha)\frac{1}{m}\ .$$

Similarly, because $\pi_{v'_1} \geq \pi_{v'_2}$ and $\pi_{v'_1} > \frac{1}{m}$, this is a contradiction.

Therefore, $m^* = m$. □