DyCause: Faster and deeper into the dynamic causalities during microservice system failures

***Abstract*—Recent methods for troubleshooting microservice system failures based on causality discovery lack the ability to construct dynamic service dependencies. This limitation, usually caused by advanced cloud system features such as auto scaling or load balance, are preventing us from identifying the accurate root cause. This paper proposes a faster and deeper analyzer for microservice system failures, named DyCause. In DyCause, we construct service dependencies using dynamic causality curves, which are aggregated from Granger causal intervals within sliding windows. Compared with traditional methods, dynamic causality curves help us construct more fine-grained dependencies with much higher efficiency in microservice system. Meanwhile, we design a backtrace searching algorithm on dependency graph to locate the root causes and rank them according to abnormality scores. Experiments on both simulated and real-world data show that DyCause outperforms several state-of-the-art methods. We further demonstrate visualizations of system failure causalities discovered from the real-world cases to show the interpretability of DyCause.**

***Index Terms*—root cause analysis, microservice system, granger causality, granger causal interval, dynamic dependency**

I. Introduction

The microservice architecture is becoming increasing prevalent in web application developments due to the high reusability, fast evolution and easy scaling features [1]. In microservice architecture, services are deconstructed into smaller granularity, such as front-end UI, messaging hub, application components and backend databases [2]. External user requests are routed through this architecture and served by dozens of different microservices. Microservice can be maintained by individual developer. They communicates with other services through lightweight web protocols and therefore can be refactored and scaled independently and dynamically [3].

Although various error protection mechanisms are introduced into modern cloud microservice system, subtle malfunctions still have chances to magnify their impact and upgrade to system failures. For instance, when a service encounters a failure and cannot return the result on time, other services that depend on it will also suffer from increased request latencies. This anomaly may propagate through the service-calling structures and eventually slow down the whole system, leading to downgraded user experiences. In commercial cloud systems, it is crucial to troubleshoot and fix the failure in quick response as massive user applications will be affected. However, the clues for defective service are hidden in the intertwined network of services. As a result, it is extremely challenging even for the knowledgable site reliability engineers (SREs) to identify the root cause of anomaly out of numerous microservices.

To help SREs diagnose failures in microservice system, many approaches [4]–[9] have been proposed recently to locate root causes from service dependency graph of the system, which is mainly constructed on performance metrics such as request latencies.

For such approaches, the accuracy of dependency graph shows the most significant impact on their performance. However, most state-of-the-art methods rely on static-graph constructing algorithms, and thus the constructed dependency graphs can not reveal the dynamic causalities during microservice system failures.

For example, Figure 1 shows a cloud application failure with dynamic causalities issues. In this example, we consider the causality between services to represent their dependencies in runtime. To build this application, the service *Cart* depends on *Ship* and *Item* while *Ship* also depends on *Item*. When service *Ship* encounters a failure that prevents it from responding correctly, service *Cart* will attempt to transparently retry connecting to service *Ship*. When the request times out, the calling from *Cart* to *Ship* stops retrying and returns an error. Because both services *Cart* and *Ship* call *Item*, their load and request latency on *Item* will increase in the same time. For the service pair *Ship* $\rightarrow$ *Cart*, as calling *Ship* results in a timeout in *Cart* during the failure, the performance metric collected from *Ship* shows weaker influence on those of *Cart*. Hence, the dependency of *Ship* $\rightarrow$ *Cart* reflected by the metric data weakens. As for *Item* $\rightarrow$ *Cart*, because they both show a trend of increasing latencies and the intermediate service *Ship* fails to bridge them, their estimated dependency will be slightly stronger. The dependencies are showed in Figure 1 (**b**). Static analysis approaches tend to build the dependency *Item* $\rightarrow$ *Cart*, while ignoring the impact of service *Ship* on this result. Therefore, in order to carry out finer-grained investigation and analysis for failure propagation, one of the critical issues is to model the dynamics of service dependencies and develop a novel analysis tool to discover such dynamics.

To address the issue caused by dynamic dependencies, this paper proposes a novel analysis approach targeting for microservice system failures, named DyCause. Contributions of this paper are listed in the following:

1) We design a novel dependency graph construction algorithm based on Granger causal intervals. In contrast to existing algorithms, we propose the concept of dynamic causality curves to model dynamic service dependencies, which helps DyCause with discovering finer-grained failure propagation processes.

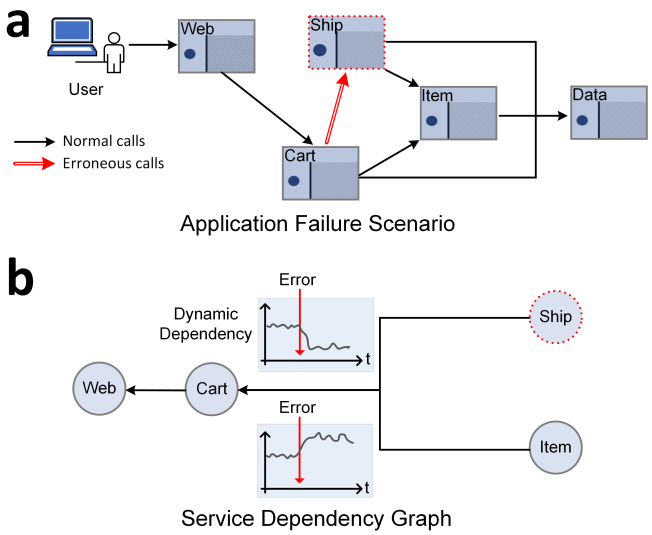

Fig. 1: (**a**) An example of application failure in a three-tier eCommerce system, where arrows represent service calls. The red colored service *Ship* is the erroneous service. (**b**) Service dependency graph of Figure 1 (**a**), where arrows represent service impacts. The plots shown in edges *Ship* $\rightarrow$ *Cart* and *Item* $\rightarrow$ *Cart* describe the dynamic dependencies.

2) We develop an optimized scheme based on Granger causal intervals to identify the root cause accurately and efficiently. With the help of the abnormal interval detection algorithm and optimized Granger causality test algorithm, DyCause outperforms other approaches even with much less metrics in terms of detection accuracy and time cost.
3) We visualize the dynamic causality curves and failure propagation chains discovered using DyCause, with cases collected from real production system. The results reflect that DyCause produces highly interpretable results, which can help SRE formulate operation and maintenance strategies.

Experiments conducted using both simulated dataset and real-world cases show that DyCause outperforms several state-of-the-art methods. Furthermore, detailed parameter analysis shows DyCause is robust to different resolutions and lengths of metric input, making our solution adaptable to various microservice system conditions.

The remainder of this paper is organized as follows. In Section 2, we summarize and categorize related works. Section 3 elaborates our solution in terms of its high-level framework and details. We present our experiments and evaluations in Section 4. In Section 5, we conclude the paper and discuss the expectations of our method and possible future works.

II. Related Work

Many previous studies [4]–[11] analyzed the problem of failure diagnosis for cloud-native system. Most of them rely heavily on the system dependency graph to guide their root cause analysis. Thus, we summarize them with respect to their dependency graph construction algorithm and show how ours differs.

In NetMedic [10], Kandula et al. leverage dependency templates to construct the dependency graph of application processes, hardware states and configuration files. Then they estimate each edge strength by comparing historical information and locate root causes by enumerating paths. NetMedic is suitable for error analysis in small enterprise systems, as limitations of templates makes it difficult to handle large and dynamic microservice systems.

Another category of approaches [4], [7] constructs dependency graph using causal discovery algorithms such as the PC algorithm [12]. For example, FacGraph [7] discovers the frequent subgraph in the dependency graphs and locate the root causes in frequent subgraphs. CloudRanger [4] develops a second-order random walk algorithm to find the root causes. Although these algorithms are data-driven, most of them assume that dependencies are static. As a result, they cannot analyze and process dynamic dependencies effectively, which will result in a decrease in the accuracy of root cause analysis.

Several approaches such as [5], [6], [8], [9] construct a call graph from service calling events. For example, TBAC [5] utilizes callers' and callees' anomaly scores in the call graph to estimate the probability of being the root causes. MonitorRank [8] uses random walk to locate root causes. Apart from above approaches, some studies combine the call graph with the graph generated by the PC algorithm to compose a two-layer dependency graph. For example, Microscope [9] and CauseInfer [6] build such two-layer dependency graphs in small enterprise applications and use a backward trace algorithm to locate root causes. On the one hand, the discovery of call graph in those methods only works best when each service resides on a specific machine. On the other hand, this type of methods rely on a monitor tool providing service calling event data. These limitations make this type of approaches much more difficult to deploy in large-scale and dynamic microservice systems. Although recent tracing tools like Canopy [13] can process numerous service calling events in large microservice systems, it takes excessive efforts to implement such tools. In contrast, our solution does not require service calling events and thus it is more system-friendly.

The concept of Granger causality [14] enables a novel way of constructing dependency graph and root cause analysis methods such as Sieve [11]. However, those methods only model static dependencies of microservice systems and can not describe the dynamics of dependencies.

In summary, most of the proposed solutions for root cause analysis in network systems are based on the assumption that the dependency between services are static. However, in modern microservice systems, various technologies such as auto scaling and load balance are constantly changing the dependencies. Based on this observation, we design a novel root cause analysis method using the notion of dynamic causality curves to subtly model the dynamic dependencies

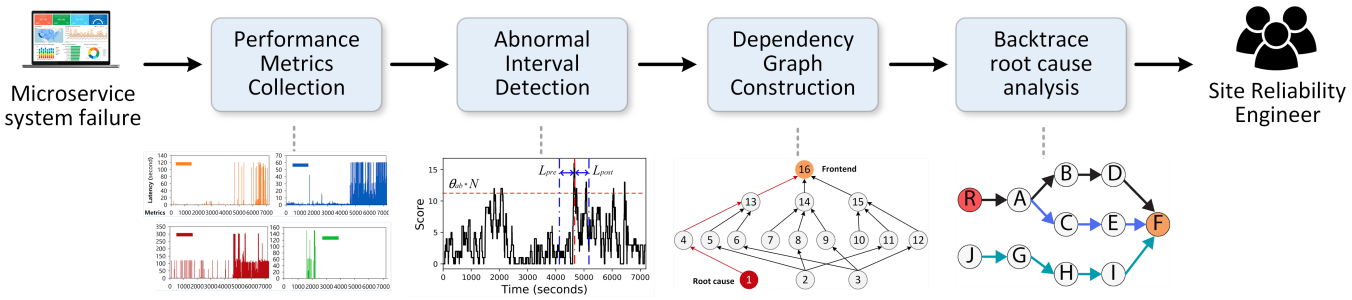

Fig. 2: Framework of DyCause.

which are ignored by most previous studies.

III. Methods

In this section, we present the details of our method—DyCause, as well as running examples.

A. Problem Definition and Overall Framework of DyCause

Generally, we formulate a microservice system as a directed graph $G(V, E, W)$, where vertices V represent services and edges E represent service dependencies of strength W. Within the system, a monitoring tool collects the performance metric data such as latencies from requests of each service, which is denoted as time series $M_i(t)$. When the system is affected by downgraded performance, we identify the problematic service v_{fe}, which is often the frontend service serving users. Our aim is to locate the root cause services responsible for the anomaly observed from service v_{fe}.

Our solution to this problem—DyCause, works in a black-box way, requiring only service performance metric, which can be easily retrieved in modern microservice systems. Figure 2 shows the overall framework of DyCause. Firstly, we collect a specific type of performance metric data as $M(t)$ during a long period. Secondly, DyCause detects the most abnormal interval within the whole period. Thirdly, the dependency graph of the microservice system is constructed based on the dynamic causality curves. Lastly, DyCause performs a backtrace search to rank possible root causes based on enumerated paths.

The following sections elaborate on each part of DyCause. Table I summarizes our major notations.

TABLE I: Notations

Notations	Definitions
$G(V, E, W)$	Service dependency graph of vertices V and impact edges E, W represents the weight of edge
N	Number of services
$M_i(t)$	Performance metric data of service i
v_{fe}	Abnormal frontend service
L_w	Size of sliding window in abnormal interval detection
λ_i	Importance level of service v_i in abnormal interval detection
L_{pre}, L_{post}	Length of pre and post part of abnormal interval
θ_{ab}	Threshold in abnormal interval detection
θ_e	Threshold in adaptive thresholding
α	Significance level in Granger causality test
L_b	Minimum step in Granger causal intervals
k	Number of selected top-k paths
n_{lead}	Number of leading services in each path
c_{path}, c_{corr}	Weights in abnormality score calculation

B. Abnormal Interval Detection

As failure may propagate through the system in a short period, we design an abnormal interval detection algorithm to detect that period before we conduct root cause analysis. Processing metrics within the abnormal interval helps DyCause focus on the error propagation, getting rid of negative impacts owing to irrelevant metrics. The abnormal interval is represented by a timestamp t_e and two corresponding period: pre-interval of length L_{pre} and post-interval of length L_{post}.

In order to accurately locate the interval when the anomaly occurs and propagates, we calculate the standard deviation $\sigma_i(t)$ using a sliding window of length L_w for each service and sum them as the system's abnormal score:

$$S_{ab}(t) = \sum_i \left(\mathbb{1}_{\{\sigma_i(t) \geq 1.0\}} \cdot \lambda_i \right) \tag{1}$$

where λ_i is the importance level of service i. If a large number of services are behaving abnormally with a standard deviation greater than 1.0, $S_{ab}(t)$ usually deviates from its normal value. When $S_{ab}(t)$ exceeds the threshold of $\theta_{ab} * N, \theta_{ab} \in (0, 1)$, we assume it represents a potential abnormal state and we choose the time with the highest score as t_e. Thus, the occurrence of the anomaly is located in the interval $[t_e - L_{pre} : t_e + L_{post}]$. Here, the parameters θ_{ab} and λ_i control the sensitivity of abnormal interval detection and L_{pre}, L_{post} determine the breadth of time considered in anomaly analysis.

In our experiments, we set θ_{ab} as 0.3 and λ_i as 1.0. We analyze the impact of parameters L_{pre}, L_{post} on the algorithm

performance through additional experiments, please refer to Section IV-D1.

We show a sample incident from our real-world testbed—a commercial cloud platform, which is a top-tier enterprise-level PaaS system. This cloud system provides over 200 categories of public cloud microservices including Big Data, Data Analytics and etc. This anomaly incident lasted for 2 hours in the cloud platform and caused the slowdown or even no response of dashboard UI services. Manual diagnosis results reveals that the root causes were several event collection services which failed to provide service. SRE team evaluates the status of microservices based on the average request latencies. They collects 33 suspicious microservices out of 1732 monitored microservices in total. The raw data recorded their API latencies per second during 2 hours.

Figure 3 plots the system's abnormal score during this incident. We highlight the detected abnormal interval with blue vertical lines and the detection threshold with red horizontal line. The system's abnormal score varies with the anomaly propagation quite sharply and forms several different peaks. Our algorithm chooses the peak around 4600 seconds where we observe the highest score. We seek advices to SREs and they confirm that this interval covers the anomaly propagation through the cloud platform. Another peak which is observed around 2000 seconds is caused by load variation of several internal security services and isn't classified as an incident as user services are not affected.

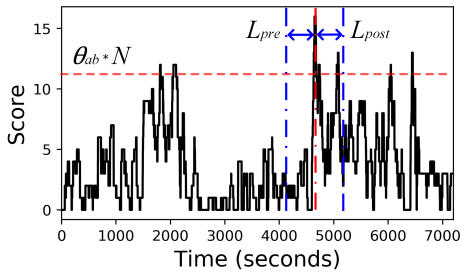

Fig. 3: The system's abnormal score in the real-world experiment. The red vertical line represents the timestamp t_e and abnormal interval is indicated by the blue vertical lines.

C. Dependency Graph Construction

After determining the abnormal interval, DyCause focuses the metrics $M^{ab}(t)$ collected within that period and constructs a dependency graph from it. Our construction algorithm examines Granger causal intervals [15] with adaptive thresholds.

1) Background on Granger causality: To illustrate our algorithm, we first introduce several key concepts of Granger causality [14] and Granger causal intervals [15].

Granger causality is defined by a hypothetical test which focuses on the relationship between two time series. In the context of this study, to discover the causality between service v_A and v_B, we first collect their performance metric data such as request latencies, denoted as $X(t), Y(t)$. Traditionally, we construct the following two linear regression models and fit them with data $X(t), Y(t)$. The only difference is that the full model has more independent variables, $\{X(t-i) \mid 1 \leq i \leq L\}$, which represent the information from service v_A.

Partial $\hat{Y}(t) = \sum_{i=1}^{L} \alpha_i^{part} Y(t-i) + b^{part}$.

Full $\hat{Y}(t) = \sum_{i=1}^{L} \alpha_i^{full} Y(t-i) + \beta_i^{full} X(t-i) + b^{full}$.

In the null hypothesis, we assume that $X(t)$ has no significant impact on $Y(t)$ and the extra information in the full model can not improve its prediction performance. As a result, statistical calculation proves that the value

$$F = \frac{(SSE_{part} - SSE_{full}) / (d_{full} - d_{part})}{SSE_{full} / (T - d_{full} - 1)} \tag{2}$$

will conform to a F-distribution of $\mathcal{F}_{d_{full}-d_{part}, T-d_{full}-1}$. Here, SSE_{part}, SSE_{full} are the sums of squared errors which are calculated using $\sum_t \left(\hat{Y}(t) - Y(t)\right)^2$ from the above two models after fitting. And d_{part}, d_{full} are the degrees of freedom of the two models, which are $L, 2L$ respectively. T is the total number of samples in time series. Therefore, we estimate the probability of the null hypothesis using Equation 2 and reject the null hypothesis if the probability is less than a small threshold, which means there exists Granger causality from $X(t)$ to $Y(t)$. We interpret the relationship as an indication that service v_A impacts service v_B.

Granger causality is generally used to test data for the entire period $[0\colon T]$. However, the causalities in microservice systems are constantly dynamic, leading to the necessity of designing a dynamic description. In view of this, we consider Granger causal intervals [15], which refer to the periods within the whole data range such as $[s_i\colon e_i]$, $0 \leq s_i < e_i \leq T$, and design a statistic tool named dynamic causality algorithm to discover temporal dynamics in microservice system failures.

2) Temporal dynamics discovery: In order to discover the temporal dynamics between services, we propose the concept of dynamic causality curve.

Definition 1. A dynamic causality curve $C_{i,j}(t)$ describes the dependency strength between service v_i and v_j over time.

To calculate $C_{i,j}(t)$, we conduct Granger causality test for each test window within the abnormal interval between service v_i and v_j. For each satisfied window $[s\colon e]$, we add 1 to the causality strength within this window. The enumeration of test windows follows the algorithm in Granger causal intervals [15] to apply its optimization techniques, as showed in Algorithm 1. Here, function $c(\alpha, L, e-s+1)$ calculates the critical value of the F-distribution for desired false-rejection probability α.

To better understand the temporal dynamics of service dependencies, we take a real-world case as example and illustrate the generated path with corresponding dynamic causality curves. As shown in Figure 4, we observe the most prominent causality periods are near the detected abnormal timestamp, reflecting the accuracy of the abnormal interval. Particularly, the curve crests (periods with highest score) of the last 3 curves (API No.27 $\rightarrow$ API No.22, API No.22 $\rightarrow$ API No.21 and API

Algorithm 1: Temporal dynamics discovery

Input: Metric data during abnormal interval $M^{ab}(t) \in \mathbb{R}^{N \times T}$, Granger causality significance level α, basic interval length L_b

Output: Dynamic causality curves $C_{i,j}(t)$.

begin
  /* Calculate dynamic causality curves from Granger causal intervals */
  for *each service pair* $v_i \rightarrow v_j$ **do**
    $C_{i,j} \longleftarrow \{0\}^T$
    for $s = 0$ to $N - L_b$ with step L_b **do**
      for $e = s + L_b$ to N with step L_b **do**
        Estimate the $\lceil F \rceil$ and $\lfloor F \rfloor$ using previous regression models
        if $\lceil F \rceil \leq c(\alpha, L, e - s + 1)$ **then**
          Continue
        if $\lfloor F \rfloor \leq c(\alpha, L, e - s + 1)$ **then**
          Conduct Granger causality test on $M_i^{ab}(s\colon e), M_j^{ab}(s\colon e)$ and calculate F
          if $F \leq c(\alpha, L, e - s + 1)$ **then**
            /* Satisfy Granger causality test */
            Increase $C_{i,j}(s\colon e)$ by 1
        else
          /* Satisfy Granger causality test */
          Increase $C_{i,j}(s\colon e)$ by 1

No.21 → API No.14) shift right in time axis, which can be explained by the failure's propagation chain of API No.27 → API No.22 → API No.21 → API No.14. If we consider the whole path, the result reveals this process (Due to double-blind review requirement, we modify and simplify specific API names): the error of event collecting service *events* (API No.28) first affected its monitoring service *abstraction* (API No.9). After that, because service *natural-language-classifier-toolkit* (API No.27) was used to classify the event text and report to security services *security-alertcon* (API No.22) and *security-news* (API No.21), they were consecutively affected. Finally, the web frontend UI service *dashboard* (API No.14) freezed.

3) Adaptive thresholding: For dynamic causality curves, we use adaptive thresholds to determine the dependency relation and strength between services, as showed in Algorithm 2. The threshold is adaptively calculated for each group of service pairs so as to make sure every significant dependency relation is included. Here, the parameter θ_e controls the sparsity of the generated dependency graph. Figure 5 shows an example of such procedure.

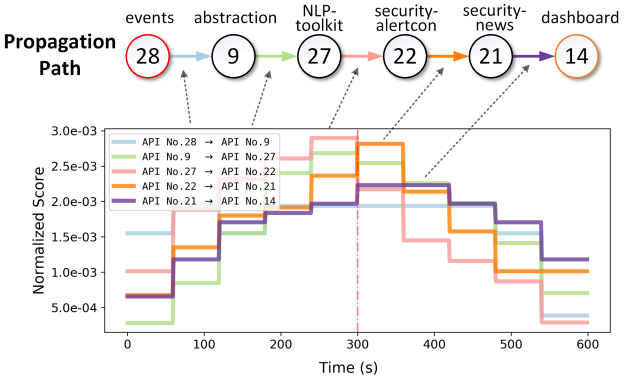

Fig. 4: Dynamic causality curves in a generated path (top) in the real-world experiment. Scores $C_{i,j}(t)$ are normalized by dividing the area under its curve. The red vertical line indicates detected abnormal timestamp.

Algorithm 2: Adaptive thresholding

Input: Dynamic causality curves $C_{i,j}(t)$, Adaptive threshold θ_e

Output: Dependency graph $G(V, E, W)$.

begin
  /* Connect edges by adaptive thresholds */
  $E \longleftarrow \emptyset$
  $W \longleftarrow \{0\}^{N \times N}$
  for *each service* i **do**
    $h \longleftarrow 0^N$
    for *each service* j **do**
      $h_j \longleftarrow \sum_t C_{i,j}(t)$
    Set edge threshold τ_i to $\theta_e * \max(h)$
    for *each service* j **do**
      if $h_j \geq \tau_i$ **then**
        $E \longleftarrow E \cup \{(v_i \rightarrow v_j)\}$
        $W_{i,j} \longleftarrow h_j / \tau_i$
  Normalize weight matrix W

D. Backtrace Root Cause Analysis

In order to perform root cause analysis with the previous dependency graph, we estimate service abnormal scores by combining two aspects of information: 1) Path correlation strength $S_{path}(v_i)$ from the abnormal frontend service; 2) Pearson correlation coefficient $S_{corr}(v_i)$.

The path correlation strength is proportional to the number of failure propagation paths starting from it, which is calculated using following steps:

Step 1 Use a backward breadth first search in the dependency graph from the frontend service v_{fe} to get all paths P_i. In order to avoid searching loops, we limit the occurrence of each service in each path to 1;

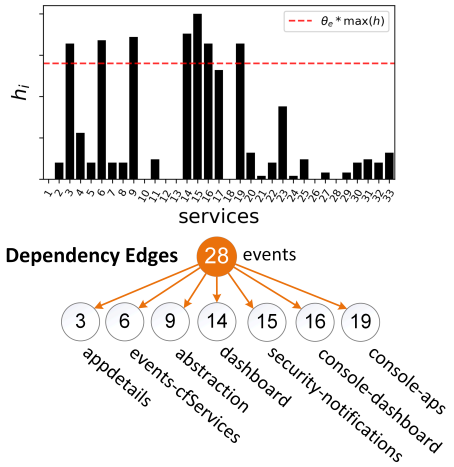

Fig. 5: Adaptive thresholds from service API No.28 and generated dependency edges in the real-world experiment. After adaptive thresholding, we found services API No.3, No.6, No.9, No.14, No.15, No.16 and No.19 depending on service API No.28.

Step 2 Estimate existence probability of each path. For path $P_i = \{i_1 \rightarrow ... \rightarrow i_n\}$ where $i_n = fe$, $Prob(P_i) = \frac{n-1}{\sum_{k=1}^{k=n-1} 1/W_{i_k,i_{k+1}}}$ where W is the weight matrix of the dependency graph;

Step 3 Count the occurrence of services in the leading part of probability ranked paths as $S_{path}(v_i)$. We select top-k paths in the ranked list to avoid counting normal services and the leading part refers to the first n_{lead} services in a path.

The backward breadth first search in DyCause helps in two ways: on the one hand, our tool can output the root cause services along with their propagation paths to help SREs understand and fix the anomaly. On the other hand, DyCause can analyze multiple anomalies propagating in composite paths, as showed in Figure 6. Here, multiple root cause services (API No.6, No.28, No.30 and No.31) cause the failure of front-end service (API No.14) through multiple paths.

The second part of the score considers the absolute Pearson correlation coefficient between the candidate service and frontend, as defined in Equation 3. This score function shows the abnormality level of the candidate service, which helps rule out many false positives that behave normally.

$$S_{corr}(v_i) = \left| \frac{\sum_k (M_i^{ab}(k) - \overline{M_i^{ab}})(M_{fe}^{ab}(k) - \overline{M_{fe}^{ab}})}{\sqrt{\sum_k (M_i^{ab}(k) - \overline{M_i^{ab}})^2} \sqrt{\sum_k (M_{fe}^{ab}(k) - \overline{M_{fe}^{ab}})^2}} \right| \quad (3)$$

The final abnormality score of each service is calculated as $c_{path} S_{path}(v_i) + c_{corr} S_{corr}(v_i)$, in which the weights c_{path}, c_{corr} are used to normalize the two scores to the same range. Finally, DyCause outputs the services ranked by their abnormality scores to SREs.

IV. Experiments

To evaluate our method, we make a comparison of root cause detection accuracy with several state-of-the-art methods. Extensive experiments are designed and conducted to show the robustness of DyCause and advantages over previous methods.

A. Datasets

Our benchmark datasets consist of dozens of incidents collected from two environments: a simulation system—Pymicro [16] and a real-world cloud platform.

Pymicro is a microservice-based simulation system containing 16 services with a pre-defined request topology as shown in Figure 7. To generate the benchmark dataset, we apply a constant load on frontend service No.16 for 1500 seconds and collect the request latencies of each service. Simultaneously, we manually increased the internal latency of service No.1 to simulate an incident. The red colored path in Figure 7 shows how the injected anomaly propagates.

In addition to the simulation system, we evaluate DyCause using real incidents collected from our cloud platform. These incidents are internally reported and manually collected from our system operation logs. The data for each incident includes about 15 million records of metrics collected from over 1000 microservices during 2 hours when it occurs. The root cause of incident is labeled by SREs based on their manual analysis results. Due to the dynamics and complexities of this enterprise cloud platform, even SREs can not determine the ground truth of dependency graph. In our experiment, part of the selected approaches [5], [8], [10] require the service call graph which is not available in our data. Thus, we borrow the state-of-the-art way leveraged in [4] to generate the required dependency graph.

B. Performance Evaluation

To evaluate the performance of DyCause, we selected several state-of-the-art approaches as benchmark, listed as follows.

- Timing Behavior Anomaly Correlation (TBAC) [5] combines service callers' and callees' scores measured by Pearson correlation coefficient to represent the abnormal level of each service. Services with highest abnormal level are considered as the root causes.
- NetMedic (NM) [10] uses pre-defined templates to construct the service dependency graph and estimate service abnormality by considering all paths leading to the affected frontend service. Services with highest abnormality are considered as the root causes.
- MonitorRank (MR) [8] performs extended random walks with self-edges and backward-edges on the service call graph to locate the root causes.
- CloudRanger (CR) [4] generates service impact graph from metrics and utilizes a second-order random walk

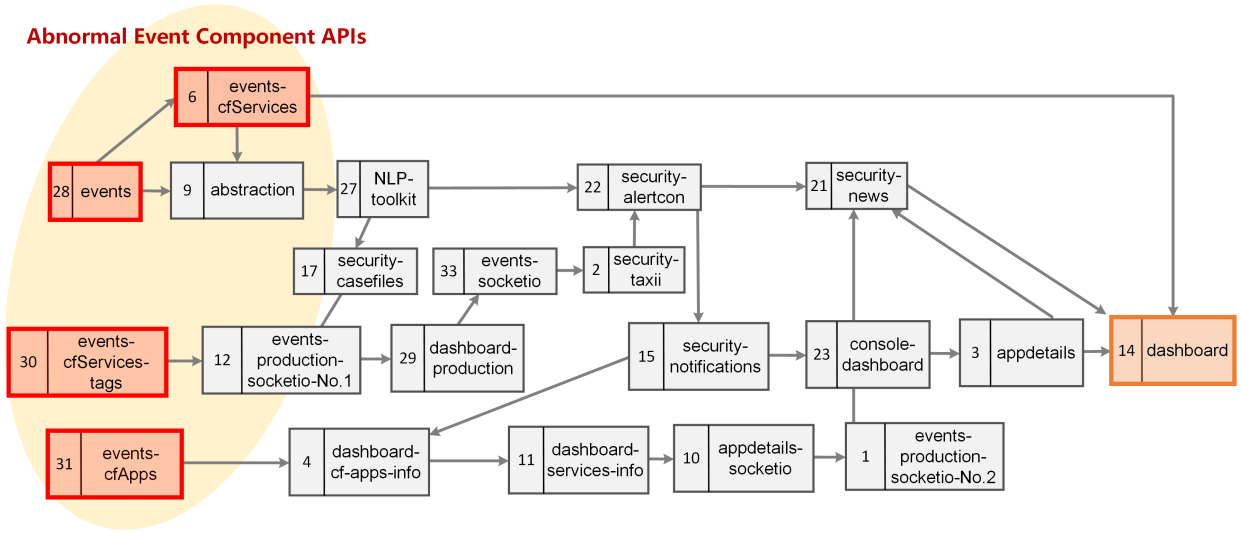

Fig. 6: Part of the dependency graph with multiple error propagation paths in our real-world experiment. Arrows represent service impacts. Red services are the root causes, yellow service is the affected frontend and others are intermediate services.

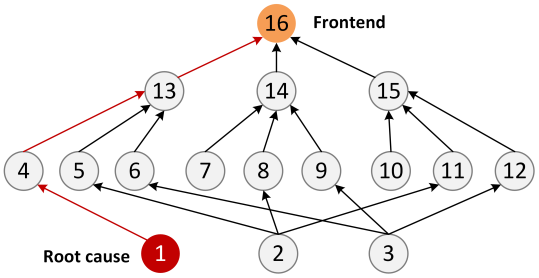

Fig. 7: The architecture of Pymicro. Arrows represent service dependencies. Service No.1 is the root cause and service No.16 is the frontend in the dataset.

algorithm which considers 1 step walking history to locate the root causes.

We introduce two performance metrics: **PR@k** and **Acc**, focusing on different aspects of root cause analysis.

Equation 4 defines **PR@k**. R^{true} is the set of ground-truth root cause services and R^{pred} is the ordered list of predicted root cause services. As **PR@k** measures how many services in the top k predicted root cause services are correct, it represents the precision of root cause analysis. In our experiments, we calculate **PR@k** with k ranging from 1 to 5 and their average **PR@Avg**.

$$\mathbf{PR@k} = \frac{\sum_{i=1}^{k} \mathbb{1}_{\{\mathrm{R}_i^{\text{pred}} \in \mathrm{R}^{\text{true}}\}}}{\min(|\mathrm{R}^{\text{true}}|, k)} \tag{4}$$

We introduce another metric **Acc** to evaluate the recall rate of root cause analysis, see Equation 5. **Acc** is calculated by averaging the prediction rank of each ground-truth root cause service. We consider the existence of multiple root cause services and let **Acc**=100% if all of them are ranked top in the predicted root cause list.

$$\mathbf{Acc} = \frac{1}{|\mathrm{R}^{\text{true}}|} \sum_{r \in \mathrm{R}^{\text{true}}} s(r)$$
$$\text{where} \quad s(r) = \begin{cases} \frac{N - \max\left(0, rank(r) - |\mathrm{R}^{\text{true}}|\right)}{N} & \text{if } r \in \mathrm{R}^{\text{pred}} \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

Experiments were conducted in a workstation equipped with CPU Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz and 256 GB RAM. The running environment is Anaconda 3 Python 3.7.

Each experiment result were averaged using 40 best parameters among 1000 runs during parameter searching. In DyCause, we use a value of 0.7 as adaptive thresholds, select 50 or 150 top-k paths where leading part size n_{lead} being 2 or 3, and set $c_{path} = \frac{1}{k}, c_{corr} = 1.0$ where k is the number of selected paths.

1) Pymicro Simulation Experiment and Analysis: As shown in Table II, our method DyCause shows the best performance for all metrics in the simulation experiment. In particular, the 100% **PR@5** of MonitorRank as well as our method indicates they both can successfully sort out the correct root cause within top 5 predictions in every test. As for **Acc**, even though both random walk methods (MonitorRank and CloudRanger) perform significantly better than path-based methods (TBAC and NetMedic). Our method DyCause outperforms them because we combine both path-based and correlation coefficient information.

The result reveals the following implications: firstly, compared with TBAC, both random walk methods (MonitorRank and CloudRanger) and path-based searching methods (NetMedic and DyCause) can help locate root causes in dependency graphs. Secondly, data-driven dependency graph construction algorithms in CloudRanger and DyCause show

advantages to the algorithm that only uses service call graphs in MonitorRank and NetMedic when finding root causes. This implies that generated graphs from the anomaly data capture more effective dependencies of the anomaly than static service call graphs. Thirdly, DyCause has higher **Acc** than CloudRanger, which demonstrates the effectiveness of dynamic causality curves in modeling service dependencies.

TABLE II: Performance on Pymicro dataset

Method	PR@1	PR@5	PR@Avg	Acc
TBAC	0.0	0.0	0.0	28.13
NM	0.0	15.79	7.37	55.76
MR	0.0	**100.0**	69.25	90.39
CR	59.0	99.0	86.94	96.33
Ours	**66.67**	**100.0**	**93.33**	**97.92**

2) Real Production Environment Experiment and Analysis: As summarized in Table III, DyCause shows the best performance except on metric **PR@1** in the real-world experiment. The **PR@1** result is caused by limited ground-truth root cause services of the real-world incident. In our real-world dataset, ground-truth root cause services are obtained by SREs' manual analysis. While in a real production enterprise system, root causes of a system failure are usually a group of services related to a malfunctioning component, instead of one particular service. In our running example analyzed in Figure 6, the event component is the root cause of the failure, and the performance of multiple related service APIs exposed by this component show abnormality. However, SREs only labeled 4 of them as the root causes (API No.6 *events-cfServices*, API No.28 *events*, API No.30 *events-cfServices-tags*, API No.31 *events-cfApps*) out of many other event-related APIs. Thus, as **PR@1** only considers the top-1 service, it is comparatively unreliable in real-world cases.

The result of this experiment shows when there are multiple root causes, the heuristic random walk algorithm in MonitorRank and CloudRanger outperforms the simple path-based algorithm in TBAC and NetMedic. Although DyCause applies a path-based searching algorithm instead of a heuristic one, we solve the problem by extending the search space in our backward breadth first search.

TABLE III: Performance on real-world dataset

Method	PR@1	PR@5	PR@Avg	Acc
TBAC	0.0	30.83	13.36	80.11
NM	0.0	57.29	20.97	90.94
MR	**95.25**	82.25	82.91	98.24
CR	92.0	88.85	84.46	97.85
Ours	78.95	**98.03**	**86.36**	**99.50**

C. Thresholding Algorithm Comparison

To verify the effectiveness of our adaptive thresholding algorithm, we implement a static thresholding algorithm, in which we accept the dependency relation when $\sum_t C_{i,j}(t)$ is larger than a static threshold θ_{static}. We conduct experiments on the real-world datasets with different thresholding algorithms, adaptive and static, the result of which is shown in Figure 8. We see that adaptive thresholds produce higher result accuracy than static thresholds and the highest **Acc** is achieved with adaptive threshold of 0.7, which indicates adaptive thresholding algorithm has higher performance and is easier to fine tune.

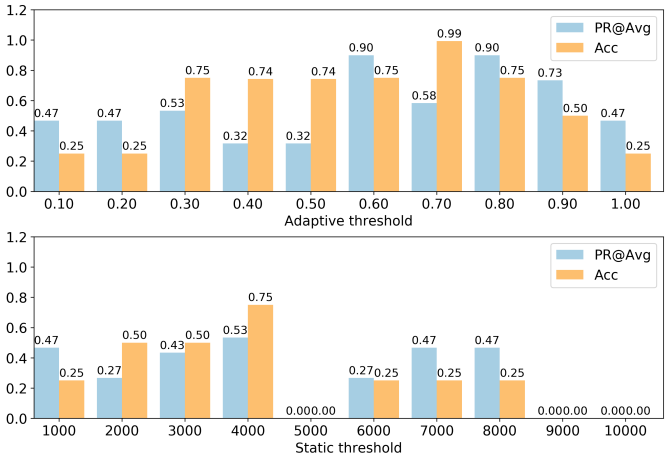

Fig. 8: Comparison of adaptive thresholds (θ_e) and static thresholds (θ_{static}) in real-world cases.

D. Data Sensitivity Analysis

To analyze the data sensitivity of DyCause, we evaluate different abnormal interval lengths, data resolutions and data lengths using real-world dataset.

1) Abnormal Interval Length: We design two sets of experiments to see how different abnormal interval lengths L_{pre}, L_{post} affect the performance, where we change the interval length while keeping the other length fixed at 0 seconds. See the results in Figure 9.

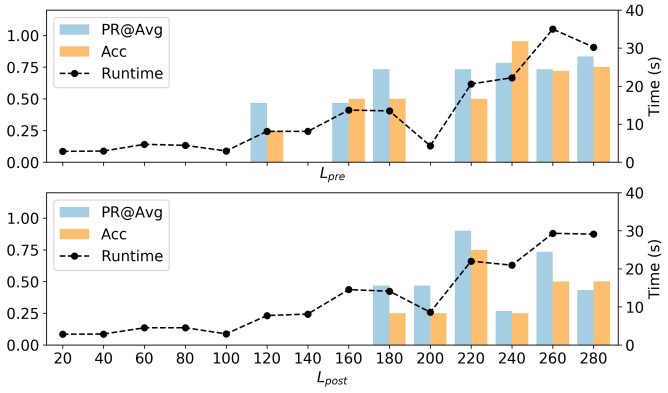

Fig. 9: Influence of pre-interval and post-interval lengths L_{pre}, L_{post} on performance and execution time of DyCause when processing real-world cases.

We find that at least 240 seconds of input data is necessary to obtain an accurate result. Considering more data does not

improve the performance significantly, which indicates that our method is of obvious advantages especially when processing limited amount of data. Moreover, in the experiment with pre-interval length, DyCause provides 95.45% **Acc**, which demonstrates that DyCause responds faster when the microservice system starts behaving abnormally. As for the execution time, the result shows that DyCause can diagnose root causes using approximately 20 seconds with 240 seconds of pre-interval length in our real-world microservices system, which is fast enough for quick response to system failures.

2) Data Resolution: To analyze the impact of data resolution (granularity) to DyCause, we aggregate the data with different resolutions and run DyCause on the sampled data. The result is summarized in Figure 10. In this evaluation, the aggregation averages data samples in the non-overlapping moving window of different length, which simulates a larger data resolution.

Figure 10 depicts that **Acc** decreases quickly to 31% when we increase the data resolution. This reflects that temporal dynamics discovery in DyCause depends on fine-grained metric data. With the help of advanced tracing infrastructures such as [13], [17], collecting such fine-grained data in large-scale microservice systems is with high feasibility. In other words, DyCause can be quickly deployed and applied in various microservice systems.

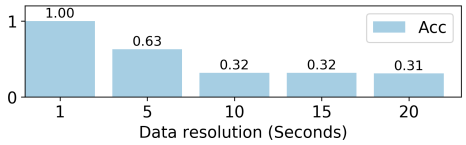

Fig. 10: Influence of data resolution on performance of DyCause in our real-world experiment.

3) Data Length: This experiment evaluates the influence of input data length. To this end, we run MonitorRank, CloudRanger and DyCause using different lengths of raw metric data. Specially, we construct the input data to cover the range in which the system failure occurs. The result of performance and time cost is shown in Figure 11.

With the increases of data length, the **Acc** for all methods increases, as well as method execution time. Moreover, DyCause runs faster than the other methods and outperforms them in accuracy as well. This experiment shows that PC algorithm in MonitorRank and CloudRanger is more time-consuming when constructing the dependency graph. While in DyCause, the abnormal interval detection algorithm helps it focus on the abnormal data, which reduces the time cost and increases the performance of DyCause.

E. Dependency Graph Visualization

To visualize DyCause and study the influence of dependency graph, we run DyCause using two different dependency graph construction algorithms: PC algorithm (PC) [12] and traditional Granger causality test algorithm (GC) [14]. The result is shown in Table IV and Figure 12.

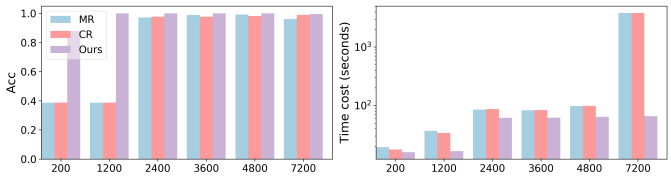

Fig. 11: Comparison of performance and time cost of MonitorRank (MR), CloudRanger (CR) and our DyCause on different data length in our real-world experiment. Here, x axis represents data length. The time cost is plotted in log scale.

First of all, the dependency graphs generated by PC algorithm with different significance value ignore the dependency edges towards the frontend service, which causes backtrace searching algorithm fail to identify any root causes. Secondly, traditional Granger causality testing tends to construct a much denser dependency graphs so that our backtrace searching algorithm has to stop searching early to avoid infinite iterations. However, such operation in backtrace searching algorithm prevents it from finding complete dependency chains, which results in affected performance. This comparison shows the importance of considering temporal dynamics when modeling service dependencies during microservice system failures.

TABLE IV: DyCause with different dependency graphs.

Graph method	PC	GC	DyCause
PR@Avg	0	60.33	86.36
Acc	0	59.85	99.50

V. Conclusion

This paper presents DyCause: a faster and deeper analyzer for microservice system failures.

In DyCause, we propose the concept of dynamic causality curves to model the dynamic service dependencies. Firstly, we apply an abnormal interval detection algorithm to locate the anomaly data for further analysis. Secondly, we generate the dynamic causality curves through Granger causality tests on many windows and subsequent aggregation. After that, we construct the dependency graph using an adaptive thresholding algorithm. Lastly, DyCause performs a backtrace searching algorithm to find possible root cause services and corresponding anomaly propagation paths. Through experiments on the simulation and real-world datasets, we show that DyCause achieves the best performance among several state-of-the-art methods.

In our experiments, DyCause shows its effectiveness and high efficiency when analyzing large-scale microservice system failures. In particular, data sensitivity experiments demonstrate that DyCause is robust, fast and easy to deploy compared to state-of-the-art approaches.

In our future work, we will devote to the visualization tool implementation to visualize the dependency graph of a microservice system in a dynamic way, which may help

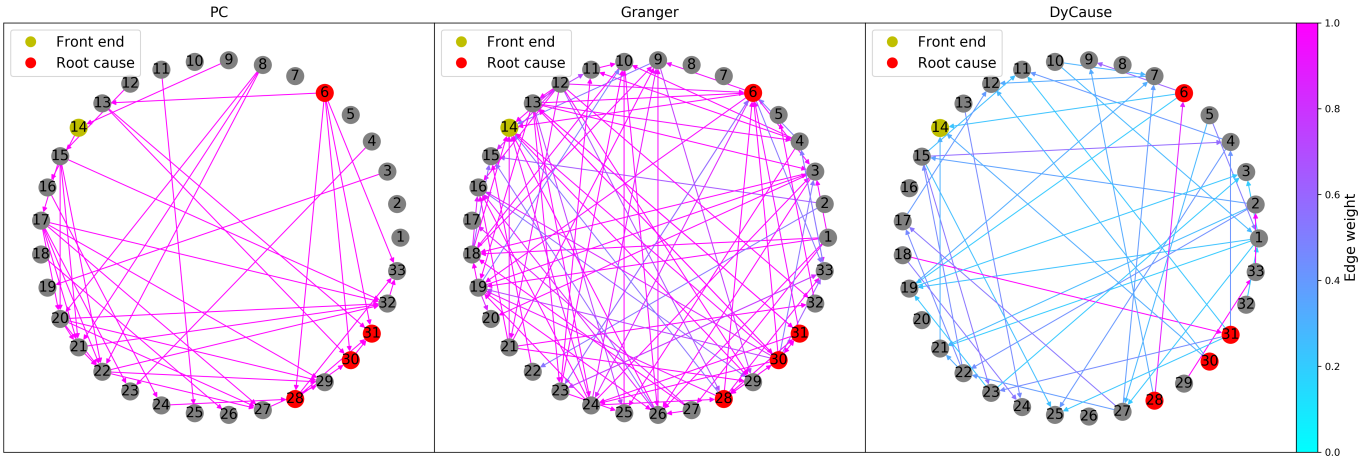

Fig. 12: Comparison of generated dependency graphs.

SREs monitor ongoing system failures. Besides, inspired by frequent subgraph mining algorithm in [7], we can analyze the frequent patterns in a series of dependency graphs from DyCause, which may reveal bottlenecks or design defects in microservice systems.

REFERENCES

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

[2] D. Oppenheimer and D. Patterson, "Architecture, operation, and dependability of large-scale internet services: three case studies," 2002.

[3] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *2015 10th Computing Colombian Conference (10CCC)*. IEEE, 2015, pp. 583–590.

[4] P. Wang, J. Xu, M. Ma, W. Lin, D. Pan, Y. Wang, and P. Chen, "Cloudranger: root cause identification for cloud native systems," in *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE Press, 2018, pp. 492–502.

[5] 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 *2009 13th European Conference on Software Maintenance and Reengineering*. IEEE, 2009, pp. 47–58.

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

[7] W. Lin, M. Ma, D. Pan, and P. Wang, "Facgraph: Frequent anomaly correlation graph mining for root cause diagnose in micro-service architecture," in *2018 IEEE 37th International Performance Computing and Communications Conference (IPCCC)*. IEEE, 2018, pp. 1–8.

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

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

[10] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl, "Detailed diagnosis in enterprise networks," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4, pp. 243–254, 2009.

[11] J. Thalheim, A. Rodrigues, I. E. Akkus, P. Bhatotia, R. Chen, B. Viswanath, L. Jiao, and C. Fetzer, "Sieve: actionable insights from monitored metrics in distributed systems," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. ACM, 2017, pp. 14–27.

[12] M. Kalisch and P. Bühlmann, "Estimating high-dimensional directed acyclic graphs with the pc-algorithm," *Journal of Machine Learning Research*, vol. 8, no. Mar, pp. 613–636, 2007.

[13] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi *et al.*, "Canopy: An end-to-end performance tracing and analysis system," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 34–50.

[14] C. W. Granger, "Investigating causal relations by econometric models and cross-spectral methods," *Econometrica: Journal of the Econometric Society*, pp. 424–438, 1969.

[15] Z. Li, G. Zheng, A. Agarwal, L. Xue, and T. Lauvaux, "Discovery of causal time intervals," in *Proceedings of the 2017 SIAM International Conference on Data Mining*. SIAM, 2017, pp. 804–812.

[16] "Pymicro," https://github.com/rshriram/pymicro, accessed: 2020-03-09.

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