TraceArk: Towards Actionable Performance Anomaly Alerting for Online Service Systems

Zhengran Zeng[‡][†][*], Yuqun Zhang[‡][*], Yong Xu[†], Minghua Ma[†],
Bo Qiao[†], Wentao Zou[§], Qingjun Chen[§], Meng Zhang[§], Xu Zhang[†],
Hongyu Zhang[¶], Xuedong Gao[§], Hao Fan[§], Saravan Rajmohan[§], Qingwei Lin[†] and Dongmei Zhang[†]
[†]Microsoft Research, {v-zhezeng, yox, minghuama, boqiao, xuzhang2, qlin, dongmeiz}@microsoft.com
[‡]Southern University of Science and Technology, {12032889, zhangyq}@sustech.edu.cn
[§]Microsoft 365, {wezo, qingche, zhangmeng, xuedong.gao, haofan, saravar}@microsoft.com
[¶]Chongqing University, hyzhang@cqu.edu.cn

***Abstract*—Performance anomaly alerting based on trace data plays an important role in assuring the quality of online service systems. However, engineers find that many anomalies reported by existing techniques are not of interest for them to take further actions. For a large scale online service with hundreds of different microservices, current methods either fire lots of false alarms by applying simple thresholds to temporal metrics (i.e., latency), or run complex end-to-end deep learning model with limited interpretability. Engineers often feel difficult to understand why anomalies are reported, which hinders the follow-up actions. In this paper, we propose an actionable anomaly alerting approach *TraceArk*. More specifically, we design an anomaly evaluation model by extracting service impact related anomalous features. A small amount of engineer experience (i.e., feedback) is also incorporated to learn the actionable anomaly alerting model. Comprehensive experiments on a real dataset of Microsoft Exchange service and an anomaly injection dataset collected from an open-source project demonstrate that *TraceArk* significantly outperforms the existing state-of-the-art approaches. The improvement in *F1* is 50.47% and 20.34% on the two datasets, respectively. Furthermore, *TraceArk* has been running stably for four months in a real production environment and showing a 2.3x improvement in *Precision* over the previous approach. *TraceArk* also provides intrepretable alerting details for engineers to take further actions.**

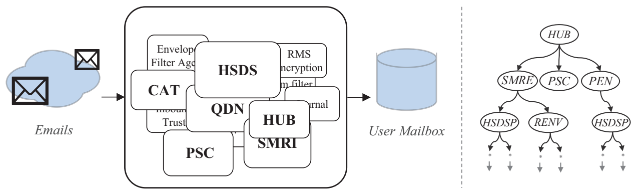
I. Introduction

Deploying software applications as online service systems, such as Microsoft Office 365, has been widely accepted by the software industry. Unlike traditional software, many online service systems are designed with a micro-service architecture and deployed on a cloud platform. However, as the service becomes more and more complex, system anomalies become inevitable and could affect user experience significantly. In this paper, we focus on performance anomalies, which are reflected by decreases in service runtime performance. Taking the email service at Microsoft as an example, there are up to 10 billion emails delivered to the cloud every day, and delivery latency is one of the critical SLAs (Service Level Agreement). Many factors, such as code bugs and hardware failures, can add to latency and cause SLA violations. Therefore, SREs (Site Reliability Engineers) employ various quality assurance techniques (i.e., performance anomaly detection and diagnosis) to reduce customer impact [1]–[6].

Timely alerting performance anomalies is crucial but challenging due to the large service scale and complexity [7]–[9]. Traces have been regarded as an important source for service understanding and performance issue diagnosis [10], [11] as they record the detailed execution flow of a request across service instances. Figure 1 shows an example of Microsoft Exchange service which is responsible for routing messages on the exchange cloud and delivering them to recipients' mailboxes. Billions of traces are generated daily with hundreds of components (names shown in abbreviated) involved to provide rich information for performance issue alerting and diagnosis. Besides shows a single trace generated from one request that involves some of the components and is visualized in a tree structure where the path represents the invocation relation. The table records the component's behavior processed from raw trace data. Each span is automatically identified by the trace system [12] and represents one microservice/function in the execution path. As shown in Figure 1, the trace structure related attributes (i.e., parentID, traceID), temporal attributes (i.e., start time, end time) and other attributes (i.e., region, type) are logged for further analysis.

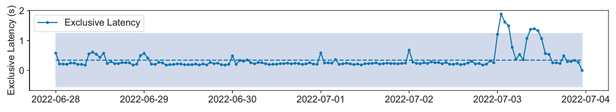
Although existing practice support fine-grained performance profiling based on system trace [11], an effective anomaly detector remains a hot topic. While the existing solutions claim high accuracy in anomaly recognition based on delicate modeling (e.g., developing advanced deep learning approaches to model trace structure [13]), the actionability of the reported anomaly is arguable. In practice, not every abnormal data pattern is investigated [14]. In industrial practice, SREs expect a list of the most influential component-level anomalies to take action, which we call "actionable anomaly alerting". Actionable means: 1) the alerted anomaly is impactful enough to motivate engineers' quick action 2) the alerting is interpretable to deliver action guidance. For example, transient spikes might be caused by system variation and not interest to engineers compared with a spike that lasts longer and may indicate a service issue. Engineers also care more about the anomalies impacting the larger service scope. Figure 2 shows a typical case in real practice. The latency of a component

[*]Equal contribution.
Work is done during Zhengran Zeng's internship at Microsoft Research. Qingwei Lin is the corresponding author.

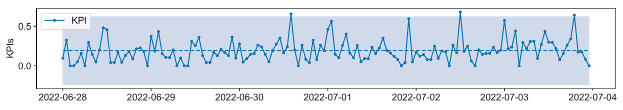


Span Id	*Span Name*	*Attributes*	*Parent Id*	*Trace Id*	*Start Time*	*End Time*
43fd263f	CAT	{region, forest, type}		43fd263f	21:49.204	21:55.225
f1f81bfe	SMRI	{region, forest, type}	76cd08c2	43fd263f	21:54.179	21:54.368
…	…	…	…	…	…	…

Fig. 1. An Example of Trace Data in an email Exchange Service



(a) Component latency result



(b) Key performance indicator result

Fig. 2. Example of non-actionable anomaly

signifies an obvious anomalous behavior of a component, while the engineers realized that there was no significant change considering the percentage of associated requests that last longer than a pre-defined threshold and thus should be marked as "non-actionable anomaly". Such feedback helps adjust the anomaly assessment model according to real scenario. Moreover, the actionability is also related to model interpretability. From our interview with SREs, they tend to apply simple anomaly alerting techniques to understand how the anomalies are alerted.

In this paper, we aim to develop an actionable anomaly alerting solution. As mentioned above, engineers' domain knowledge should be incorporated when modeling anomalous trace patterns to fit the scenario need. In a microservice system with hundreds of different components deployed in various scales (from several servers to worldwide), it is not practical to manually set a proper threshold as rules for all situations. Some approaches monitor and fire alerts on end-to-end service latency (i.e., 95th, 99th percentile) first, then gradually pinpoint the anomalous components [10], [11]. However, this would postpone the alerting of a regressed component. Recent studies tried to pursue high alerting accuracy by modeling the component behavior using embedding of the fine-grained trace structure and dynamics of whole service [15], [16]. However, the complexity of these models makes them hard to be applied to the large-scale online service system. The interpretability of "interpretable" models proposed by previous work is questionable for the developers to understand the alerting process, and thus hinders the follow-up action. Existing work also tried to classify the anomaly severity but relies on historical labels, which are hard to obtain in practice [14].

To address the above issues, we propose *TraceArk*, an actionable anomaly alerting method. First, we design an anomaly assessment method that considers both the trace structure characteristics based on an empirical study of real trace data and the engineers' perspective on the anomaly. Then, inspired by the study on active anomaly discovery [17], [18], we incorporate a small amount of engineer experience (i.e., feedback) to learn the alerting decision, which enables *TraceArk* to adapt to different anomalous scenarios. Meanwhile, we infuse interpretability to *TraceArk* in both the model and alert report. We adopt a tree-based model to keep the anomaly assessment understandable and also report the impact path of anomalous components for engineers' action guidance, i.e., the engineer can trace the path to further diagnose the anomaly. The detail discussion is provided at section V-B. To evaluate the effectiveness of *TraceArk*, we have conducted comprehensive experiments on a real dataset of Exchange services and an anomaly injection dataset collected from an open-source project. The results show that *TraceArk* significantly outperforms the existing state-of-the-art approaches. Furthermore, we have deployed *TraceArk* in a real production environment for Microsoft Exchange service. *TraceArk* has been running stably for four months, and the evaluation showed that *TraceArk* indeed facilitates efficient performance anomaly alerting and benefits further mitigation actions.

Overall, our main contributions of this study are as follows:

- We present *TraceArk*, an effective anomaly alerting approach which incorporates engineers' perspective on service performance anomaly and enables actionable anomaly alerting for microservice.
- We conduct comprehensive experiments on a real dataset of the Exchange service and an anomaly injection dataset based on an open-source project. The results show that *TraceArk* outperforms previous state-of-the-art baselines by 50.47% in terms of $F1$ under real industrial dataset, and also reach a 92% $F1$ score when getting 30 feedback samples in the synthetic dataset.
- We have deployed *TraceArk* on Exchange service in a real production scenario. *TraceArk* achieves a $Precision$ of 0.9068, which is 2.38X better than previous approaches. Moreover, the interpretability of *TraceArk* significantly benefit engineers' further actions.

II. Background and Motivation

A. Service monitoring and anomaly alerting in industry

Here we use service Exchange as an example to introduce performance monitoring and anomaly alerting in practice. Exchange is responsible for routing messages on the exchange cloud and delivering them to recipients' mailboxes under SLA (service level agreement). Messages could be routed through multiple servers in a complex flow and through several hundred components. Each component adds some delays to the message and impacts the end-to-end delivery latency.

In the Exchange team, a major task SREs undertake is to timely catch when there is a performance issue with a

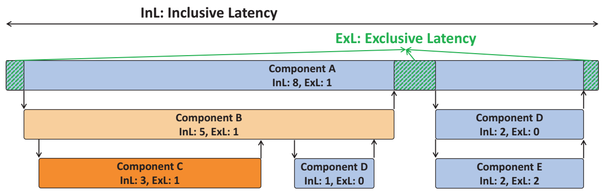


Fig. 3. Example of trace

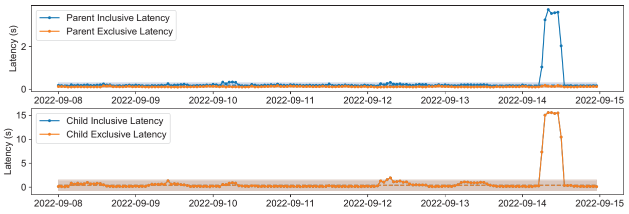


Fig. 4. Latency of a parent component and its child component

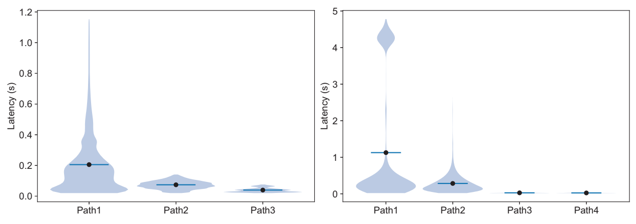
component. The latency of each message and component taken during the mail flow is traced and stored in Azure storage for further analysis. A pipeline is built to process the trace data and automatically fire alerts to corresponding SREs. Before applying *TraceArk*, Exchange built some monitors that track the latency time-series data of some key components each with their own thresholds. The monitors contain lots of false alarms, and SREs cannot investigate all the alerts during their rotation. The monitors are also not smart enough to catch all the component issues, and SREs sometimes only notice the issue until there is a big customer impact. This motivates us to design an actionable anomaly alerting solution that incorporates both the trace structure characteristics and engineers' perspectives when modeling the component behavior. We leave more details in the following sections.

B. Challenges

In this paper, we aim to develop an anomaly alerting solution that is adaptive to each microservice component and highly actionable for SREs. There are some key challenges: 1) Huge trace data volume. As mentioned, billions of traces of email requests are generated daily, which is time and space expensive to process. 2) The incorporation of SREs' knowledge. Designing an effective and efficient feedback strategy to incorporate engineers' knowledge without much effort is not trivial. 3) Noise. In the production environment, latencies are not stable for many components, and there are always duplicated alerts due to structural dependency. For example, a regression on the child component may propagate the anomaly to its parent and even parents of the parent, which makes the alert confused to engineers. 4) Model interpretablility. To make the anomaly alerting actionable, the model should be understandable to engineers without scarifying effectiveness.

C. Insights

Before proposing our solution of anomaly alerting, it is helpful to understand real trace data and anomalies. To this



(a) Distribution of component A (b) Distribution of component B

Fig. 5. Latency distribution of same component in different path, where the black point indicates the mean latency

end, we first collect extensive trace data from the Exchange service. Each trace entry records the execution flow of a customer request, which contains the span name, span invocation information, and the start and end time.

Based on such information, we can calculate the Inclusive Latency (*InL*) data of each component in the trace (i.e., individual service components), which is the subtraction of the component entry and exit time, and the Exclusive Latency (*ExL*) data, which is the running time of the component itself. They are two mainstream trace latency definitions and have been widely adopted by previous anomaly alerting studies [11], [13], [15]. In general, component inclusive latency is equal to the sum of the component exclusive latency itself and the inclusive latency of its children, e.g., Component A *InL* is equal to Component A *ExL* + Component B *InL* + Component D *InL* as demonstrated in Figure 3. As discussed in the work on tprof [11], performing anomaly alerting based on the *ExL* metric (i.e., *operation_self* in tprof) would be a better choice since it can eliminate the abnormal effects of child components on the parent components and thus precisely locate the truly anomalous components. For instance, Component C in Figure 3 is abnormal, which makes its parent B also anomalous in the *InL* metric. A more detailed real-life example is shown in Figure 4, where the child service shows an anomaly in both *ExL* and *InL*, which results in the parent component also showing an anomaly in InL. However, the parent is not a true anomaly since its *ExL* metric is healthy. Therefore, *TraceArk* mainly focuses on *ExL* metric as it can remove the anomaly noise.

Insight 1: Exclusive latency can eliminate the anomaly noise and better serve anomaly alerting.

Next, we further investigate the latency distribution of the same component in different trace paths. As discussed by previous studies [11], [13], [15], the same trace event could have different latency distributions under different trace paths. For instance, Component D in Figure 3 has an average inclusive latency of 2 seconds under path *A>D* However, its average inclusive latency in another path *A>B>D* is 1 second. And this happens in real trace data as well. Figure 5 illustrates the instance latency distribution of the same microservice component under different invocation paths. Specifically, component

A has three different invocation paths that end with component A, and their latency distributions are significantly different, i.e., 0.27s, 0.10s and 0.04s respectively. Meanwhile, component B also shows a different latency distribution under four different paths. Moreover, component B shows an anomaly with latency larger than 4 seconds under path1, indicating that the anomaly could happen in one specific invocation path and not in another. Therefore, the same trace event under different trace paths could have different latency distributions and even different running states (i.e., normal/abnormal). This may due to the fact that the same trace event under different trace paths represents the trace event processing different types of requests, e.g., a spam filtering service will have different overheads when processing emails with and without images, and anomalies may arise when processing emails with images. Besides, according to previous research [11], hierarchically aggregated traces at each layer of the hierarchy help to diagnose anomalies. We, therefore, aggregate trace at three horizontal granularities, i.e., Service level, Operation level and Path level similar to previous work [11]. Besides, we do not adopt the tree granularity in previous work, because traces in the practical online service system have a rich tree structure, and majority of traces have a unique tree structure.

Insight 2: The latency distributions of a specific component could vary under different trace paths. Anomaly assessment should incorporate fine-grained trace structure information.

Lastly, as mentioned earlier, engineers have different preferences for different anomaly manifestations. They tend to pay additional attention to anomalies that last longer or have a larger impact. As an anomaly example in Figure 4, this anomaly lasts for about seven hours, and engineers consider it to be a rather serious anomaly and therefore deserves further investigation. In contrast, for the spike anomaly that duration less than one hour may seem less important to the engineer. Besides, anomaly duration is only one of the indicators that engineers use to determine whether an anomaly is worth further processing. In practice, their judgment of an actionable anomaly is based on many more dimensions, such as the value of the anomaly delay, the region where the anomaly arises, and the scope of the impact, etc. Therefore it is difficult to model this knowledge with explicit rules. Moreover, such knowledge does not exist in raw trace data, and therefore hard to be obtained by unsupervised methods. To this end, we extract multiple features of anomalies and introduce a feedback mechanism to model engineers' knowledge of actionable anomalies from a small number of feedback samples using an interpretable machine learning model.

Insight 3: We need the knowledge of engineers to determine whether anomalies are worthy of alerting, and such knowledge is difficult to model by simple threshold rules and unsupervised methods.

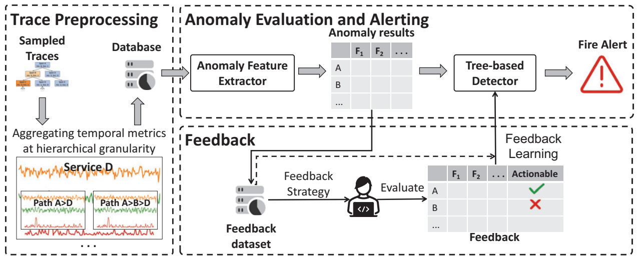


Fig. 6. The overall architecture of *TraceArk*

III. Proposed Approach

Figure 6 presents the overall architecture of *TraceArk*, which consists of ***Trace Preprocessing***, ***Anomaly Evaluation and Alerting***, and ***Feedback***. Trace Preprocessing phase samples and aggregates massive trace data considering the structure characteristics and other service related properties as studied above. Anomaly Evaluation and Alerting phase is designed for extracting and computing anomalous features of the alerting subjects, and finally to alert actionable anomalies. The Feedback module incorporates engineers' feedback on alerted anomaly and thus could dynamically adjust the model parameters to suit different anomaly preferences.

A. Trace Preprocessing

An online service system can generate a tremendous amount of traces, and these traces are automatically collected and stored in a large database by the tracing system. However, it is impractical to process all the trace data every day. Therefore, before pre-processing, we perform unbiased sampling of the trace data (i.e., randomly sampling 1% of the trace data and each trace includes the complete components invocation path) to obtain an unbiased state of the system [19] while keeping the pre-processing overhead in a desirable level (i.e., about 1 hour to process each day's data).

Then, while pre-processing the post-sampling trace data, *TraceArk* mainly focuses on the *ExL* metric, which can eliminate the abnormal impact of child components on the parent components and thus precisely locate the truly anomalous components, as discussed in Insight 1. Next, we capture the temporal behavior of the component by aggregating the time series in terms of component invocation paths. For example, six components in Figure 3 can generate six path records (i.e., A, $A{>}B$, $A{>}D$, $A{>}B{>}C$, $A{>}B{>}D$, $A{>}D{>}E$). Meanwhile, according to previous research [11] and Insight 2, different granularities of anomaly alerting results benefit the actionability. We, therefore, record those fine-grain metadata for each component, such as the **Operation** type (the type of request, e.g., POST and GET of the HTTP request), **Service** name (i.e., name of the span that the component belongs to), to facilitate the retrieval of different granular data from the database. Besides, we also collect more performance-related time series indicators such as the *InL* metric (because it is useful to indicate anomaly propagation between parents and children components), 95th percentile *ExL* metric and 99th percentile

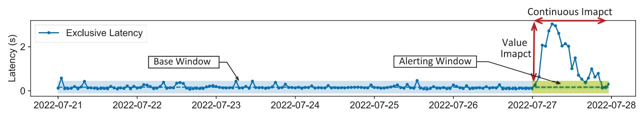


Fig. 7. Example of aggregated time-series data

ExL metric to provide engineers with more information for anomaly diagnosis.

B. Anomaly Evaluation and Alerting

Once the trace is pre-processed, all the data required for anomaly alerting will be stored in the dataset. We can easily extract anomaly features at different levels (i.e., **Service**, **Operation**, **Path**) and on different time-series metrics (e.g., *ExL* indicator, *InL* indicator) using the *Anomaly Feature Extractor* component in Figure 6. For instance, we take **Path** as the granularity for alerting anomalies based on the *ExL* indicator. First, we set the alerting window according to the practical requirement (e.g., *Day T 00:00:00* to *Day T 23:59:59*) and retrieve the time-series data within the alerting window in **Path** granularity from the database (name it alerting data). Then the data before the alerting window (e.g., one week from *Day T-7 00:00:00* to *Day T 00:00:00*) is selected (name it base data) because we believe that these data reflect the latency distribution pattern of components in the general situation (i.e., without anomaly). Finally, the data are re-aggregated (e.g., into hourly average time-series data) to further remove minor noise. Specifically, Figure 7 presents one aggregated anomaly example in which each data point indicates the average *ExL* in one hour and we denote it as *value*. Meanwhile, we also record the total count of this alerting subject appearing in all traces data within each hour as *subject_count*, and record the total count of traces within each hour as *trace_count*.

$$ContinueNum_i = \begin{cases} ContinueNum_{i-1} + 1 & \text{if } diff_i > 0 \\ 0 & \text{if } diff_i \leq 0 \text{ or Initialization} \end{cases} \tag{1}$$

$$ValueScore = \sum_i^{diff_i > 0} \frac{diff_i}{k \times \sigma} \tag{2}$$

$$ContinueScore = \sum_i^{diff_i > 0} ContinueNum_i^2 \tag{3}$$

$$TrendScore = \sum_i^{diff_i > 0} ContinueNum_i^2 \times \frac{diff_i}{k \times \sigma} \tag{4}$$

$$NormalizedCount = \frac{\sum subject\_count_i}{\sum trace\_count_i} \tag{5}$$

$$Overhead = \begin{matrix} NormalizedCount \times \\ (avg(AlertingValue) - avg(BaseValue)) \end{matrix} \tag{6}$$

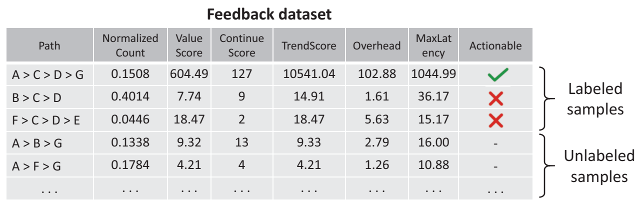
After obtaining the required time-series data, we can calculate the anomaly score for each alerting subject, which considers both the numerical changes of the anomaly and the continuity of the anomaly (since a persistent anomaly have higher priority than a temporary one). As shown in Figure 7, *ValueScore* [20] in Equation 2 is the numerical changes of the anomaly where *diff* denotes the deviation of the *value* from the K-sigma threshold. *ContinueScore* in Equation 3 represents the continuity of the anomaly which is the sum of *ContinueNum*, and *ContinueNum* indicates the number of continuous points that exceeded the health threshold. *TrendScore* in Equation 4 takes into account both the numerical changes of the anomaly and the anomaly continuity which is a more comprehensive indicator.

The *TrendScore* actually indicates the anomalous degree in temporal and can be used to perform anomaly alerting by an anomaly threshold. However, estimating this anomaly threshold is tricky, and engineers are usually concerned about the service impact when prioritizing anomaly alerts [14], [17]. To this end, we consider providing more helpful anomaly features for the threshold estimation method in addition to the above three temporal features. Therefore, based on previous empirical studies [11], [15] and engineers' experience, we decide to extract features such as the *NormalizeCount* of the alerting subject (i.e., the number of appearances in the alerting window and normalized by the number of all traces as present in Equation 5), its *MaxLatency* (i.e., maximum latency value $max(value)$) and *Overhead* (i.e., the increase in the total latency of the alerting window relative to the base window data as present in Equation 6) as the basis for assessing whether the alerting subject is an actionable anomaly before asking for further feedback from engineers. Our subsequent experiments demonstrated the effectiveness of these features.

C. Feedback

As mentioned in the Introduction, previous studies [13], [15], [18] try to model the component behavior based on latency data and structure information, but they overlook the actionability of alerted anomaly. In practice, the output of most previous anomaly alerting algorithms was anomaly ranking or a fixed number of anomaly items [11], which implies that their anomaly thresholds need to be manually adjusted. Determining anomaly threshold is a critical issue [17], as different projects or even different engineers could have different preferences, e.g., some anomalies are essential and actionable under a specific project, while similar anomalies may seem trivial to the engineers of another project. An over-extended anomaly threshold would result in under-reporting, i.e., low recall, while an under-extended threshold will result in a false alarm, i.e., low precision. Therefore, the determination of the anomaly threshold is a key factor affecting the performance of an anomaly alerting algorithm. Previous studies [17], [18] have devoted substantial effort to this issue and found that engineers are willing to mark a small number of samples to yield better alerting results. Inspired by these studies, we also designed the feedback mechanism in *TraceArk* to incorporate a small amount of engineers' feedback to learn actionable anomaly thresholds.

To this end, we chose the highly interpretable tree-based machine learning model XGBoost [21] and the semi-supervised learning paradigm [22]. When new anomaly results (initially there are some anomalies identified by simple pre-defined threshold rules) are produced after the *Anomaly Evaluation and Alerting* phase, we add them to a Feedback dataset, as demonstrated in Figure 8. Engineers can then fetch several unlabeled samples (i.e., alerting subjects) from it (the fetched samples will be given by a non-playback selection strategy as



Feedback dataset

Path	Normalized Count	Value Score	Continue Score	TrendScore	Overhead	MaxLat ency	Actionable
A > C > D > G	0.1508	604.49	127	10541.04	102.88	1044.99	✓
B > C > D	0.4014	7.74	9	14.91	1.61	36.17	✗
F > C > D > E	0.0446	18.47	2	18.47	5.63	15.17	✗
A > B > G	0.1338	9.32	13	9.33	2.79	16.00	-
A > F > G	0.1784	4.21	4	4.21	1.26	10.88	-
. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .

Labeled samples
Unlabeled samples

Fig. 8. Example of feedback dataset

further described in Section IV-C1). After the engineer finishes labeling the anomalies the labeled samples and all the samples in the Feedback database are fed into the XGBoost model for semi-supervised learning to learn and update the anomaly thresholds derived from the parameters of the model. Finally, the samples identified as anomalies will be reported to the engineers. Moreover, as the new samples coming, the outdated ones (e.g., samples of the previous month) will be removed from Feedback dataset to enable the model adaptive to the update of online service systems.

IV. Evaluation

This section presents the effectiveness of *TraceArk*. We conduct comprehensive experiments on two representative datasets and several related state-of-the-art (SOTA) baselines to answer the following research questions:

- **RQ1:** *How effective is TraceArk compared with baseline approaches?* For this research question, we extensively study the effectiveness of online service performance anomaly alerting between *TraceArk* and selected baselines on two representative datasets.
- **RQ2:** *How the feedback module improves TraceArk effectiveness?* In this research question, we delve into the performance of different feedback settings to demonstrate the effectiveness of the feedback module.

A. Experimental Setup

1) Benchmark System and Dataset: To evaluate the effectiveness and efficiency of *TraceArk*, we collected two representative trace datasets of online service systems. One dataset is collected from a real Microsoft online service system. Also, for a comprehensive comparison with previous studies [11], [13], [15], we collected a dataset with labels obtained by performing fault injection on TrainTicket [23], an open-source microservice system designed for research.

- **Exchange**: A real industrial trace dataset collected from Microsoft M365 Exchange service. When the user sends an email, the email passes through different microservice components and leaves a log record in the logging system. The real-world system may have hundreds of millions of data per day, and the experiment dataset is processed as discussed in Section III-A. Table I illustrates the statistics of the Exchange dataset, which contains several millions traces collected across dozens of days. Meanwhile, we build this dataset based on the practical scenario of the Exchange service. In this scenario, we set the anomaly alerting window to one-day for engineers' daily troubleshooting work, and set the data aggregation unit (i.e., time-series unit) to one hour, and selected the data of the past six days as base data. In the end, we collected a dataset with 8074 alerting subjects in Path granularity, and worked with experienced Exchange engineers to identify and label 439 actionable exceptions in this dataset. Note that Exchange dataset does not have Operation as the trace log do not contain this metadata, thus we ignore this granularity in the subsequent experiments.
- **TrainTicket**: TrainTicket is a train ticket booking system based on microservice architecture, which contains 41 microservices and has been widely investigated in previous microservice anomaly detection studies [11], [24]. Therefore, to compare *TraceArk* with previous baselines, we further performed fault injection on TrainTicket following previous research [25] and gathered a labeled dataset. In particular, we first deployed the latest version of the TrainTicket service on the kubernetes [26] cluster of our local machine. Then, chaos-mesh [27] (a powerful chaos engineering platform for kubernetes) injects anomalies into TrainTicket, e.g., network faults, pod faults, and stress scenarios. Similar to previous research [25], we injected six different kinds of anomalies into TrainTicket (including CPU/memory stress on pods, pod failure, and packet corrupt/loss/delay on pods) at random locations and time frames. Consistent with the previous study [15], [25], we added periodicity to the workload to simulate periodicity in real-world service systems. Finally, Jaeger [12] (an open-source, end-to-end tracing system) is employed to record the trace data generated by TrainTicket. Table I presents the statistics of the TrainTicket dataset, which contains ten days of 1,085,719 trace data. Meanwhile, due to the slight fluctuation of trace latency in TrainTicket dataset, we set the alerting window to half an hour, set the data aggregation unit to 1 minute, and selected the data of the past half hour as base data. Consequently, we collected a dataset with 19,285 alerting subjects in three granularities. Meanwhile, by giving the time and locations of faults injection, we can get the exact anomaly labels by the injection time and duration and finally identify and label 354 anomalies in three granularities.

2) Baselines: We compare *TraceArk* with the following state-of-the-art trace-based and monitoring-based anomaly alerting approaches [28]. In particular, trace-based approaches detect the anomalies by analyzing the differences between normal traces and the traces to be tested with respect to trace latency or trace structure, while monitoring-based approaches directly monitor the KPI time-series data of each alerting subject.

Trace-based anomaly alerting techniques:

- ***Tprof*** [11] is a performance profiling approach which

TABLE I
DATASET STATISTICS

Dataset	#Traces	Time	#Microservice spans	Anomaly Source	Aggregation Unit	Alerting Window	Base Window	#Alerting Subjects			#Anomalies		
								Service	Operation	Path	Service	Operation	Path
Exchange	several million	dozens of days	several hundred	Real-world anomaly	1 hour	1 day	6 day	3212	/	8074	171	/	268
TrainTicket	1085719	10 day	41	Injected by us	1 minute	30 minute	30 minute	2185	5890	11210	64	105	185

utilizes distributed tracing systems to collect trace data and then gradually identify performance issues in a hierarchical manner.

- ***TraceAnomaly*** [15] is a trace anomaly alerting approach with call-path-based trace representation and a posterior flow deep bayesian network.
- ***MultiModalTrace*** [13] is a trace anomaly alerting approach, which utilizes bimodal distributed trace information with trace latency and trace structure.

Monitoring-based anomaly alerting techniques:

- ***AutoEncoder*** [29] is a time-series anomaly alerting approach, which utilizes neural networks to calculate the reconstruction errors of data and detect anomaly.
- ***DeepSVDD*** [30] trains a neural network by minimizing the volume of a hypersphere that encloses the network representations of the data, and then calculates the anomaly score based on the distance from the center of data points.
- ***ECOD*** [31] is a unsupervised outlier alerting algorithm based on empirical cumulative distribution functions [31].
- ***IForest*** [32] monitor outliers in time-series data by building an isolation forest.
- ***KNN*** [33] is a time-series anomaly alerting approach using the distance to the *k-th* nearest neighbor of each data point as the outlier score.
- ***LUNAR*** [34] predicts the anomaly score of each data point by feeding its ordered list of distances to its *k* nearest neighbours to a neural network.
- ***SUOD*** [35] is an ensemble model that incorporates multiple anomaly alerting models.

Note that log-based anomaly alerting methods are also promising [24], [28], [36], [37], however, in this work we focus on performance anomaly alerting based on trace data.

3) Metrics: Following previous research [13], [15], we employed *Precision*, *Recall*, and *F1* to evaluate the effectiveness of anomaly alerting approaches based on *TP* (True Positive), *FP* (False Positive), and *FN* (False Negative).

- ***Precision*:** the fraction of anomaly instances among all predicted anomaly instances, i.e., $Precision = \frac{TP}{TP+FP}$.
- ***Recall*:** the fraction of anomaly instances that were predicted as anomaly, i.e., $Recall = \frac{TP}{TP+FN}$.
- ***F1*:** the harmonic mean of the *Precision* and *Recall*, i.e., $F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$.
- ***AUC*:** a ranking-based metric which tests whether positives are ranked higher than negatives.

Notice that calculating of *Precision* and *Recall* metrics requires to defining thresholds, and different thresholds can significantly affect the evaluation results. Therefore, referring to the previous study [18], we adopt the optimal threshold for each baseline to calculate the best existing F1 (denoted as $F1_{best}$), as well as the corresponding precision and recall. Besides, we further introduce a ranking-based and threshold-free metric, *AUC*, which evaluates whether positives are ranked higher than negatives.

4) Implementation: The implementation of *TraceArk* is based on Python and the XGBoost package. We set *k* to 3 at Equation 4 by following the widely adopted 3-sigma principles [38], and set the recommendation count at Section IV-C1 to 10. We set the sub-tree number of *XGBoost* to 3, while all other parameters of the *XGBoost* model are in default settings. We directly adopted the original code released by *Tprof* and *TraceAnomaly* in our experiments. For *MultiModalTrace*, which is not open sourced, we implemented it based on their papers. We implemented the rest of the monitoring-based anomaly alerting baselines using pyod [39] (a comprehensive and scalable python library for outlier/anomaly alerting).

All the experiments are conducted on a Ubuntu 18.04 server with Intel Xeon Gold 6140 CPU, 768 GB RAM, 2 Nvidia Titan-V 12GB GPU.

B. Effectiveness Evaluation

In our experiment, there are 11286 alerting subjects in the Exchange dataset and 19285 alerting subjects in the TrainTicket dataset. We first demonstrate the detailed effectiveness evaluation results of *TraceArk* without feedback mechanism and leave the evaluation of feedback module in next section. The result together with baselines on these two datasets are presented in Table II and Table III, respectively.

Table II presents the effectiveness evaluation results of *TraceArk* and other baselines on the TrainTicket dataset under different granularities. The results demonstrate that *TraceArk* significantly outperforms other baselines even without feedback tuning (i.e., detecting anomaly based on *TrendScore*) and achieves the best *AUC* and *F1* at all alerting granularities. Specifically, *TraceArk* achieves the best *F1* at the fine-grained Path granularity, with a 34.90% improvement over the best trace-based anomaly alerting approach (i.e., 0.9373 of *TraceArk* v.s. 0.6948 of *MultiModalTrace*). *MultiModalTrace* achieves similar *Recall* as *TraceArk*, while its *Precision* stands low at 0.5527, compared to *TraceArk*'s 0.9338, indicating that *MultiModalTrace* has more false positives. The reason is that the trace-based methods usually design one general end-to-end anomaly assessment model and then apply it to different components, which, however, may not perform well since the behavior of different components actually varies. In contrast, *TraceArk* has a training phase for each microservice

TABLE II
EFFECTIVENESS EVALUATION OF THE TRAINTICKET DATASET UNDER DIFFERENT GRANULARITY

TrainTicket	Service				Operation				Path			
	AUC	*Precision*	*Recall*	$F1_{best}$	*AUC*	*Precision*	*Recall*	$F1_{best}$	*AUC*	*Precision*	*Recall*	$F1_{best}$
Tprof	/	.1591	.8116	.2660	/	.1674	.6857	.2692	/	/	/	/
TraceAnomaly	.9250	**1.000**	.7246	.8403	.9077	**1.000**	.4286	.6000	.7555	.1958	.1514	.1707
MultiModalTrace	.9987	.8961	**1.000**	.9452	.9937	.5642	**.9619**	.7113	.9942	.5527	.9351	.6948
AutoEncoder	.9966	.8205	.9275	.8707	.9964	.7227	.8190	.7679	.9969	.6695	.8889	.7637
DeepSVDD	.9968	.8400	.9130	.8750	.9964	.6765	.8762	.7635	.9970	.7347	.8000	.7660
ECOD	.8799	.5763	.4928	.5313	.8831	.3478	.3810	.3636	.8233	.2906	.3278	.3081
IForest	.8507	.1517	.4638	.2286	.8766	.1312	.5048	.2083	.8393	.1176	.3222	.1724
KNN	.9966	.8955	.8696	.8824	.9924	.5641	.8381	.6743	.9933	.5470	.8722	.6724
LUNAR	.9971	.8824	.8696	.8759	.9968	.7778	.8000	.7887	.9973	.7110	.8611	.7789
SUOD	.7816	.1348	.3478	.1943	.7829	.0800	.3429	.1297	.7437	.0655	.1500	.0912
TraceArk	**.9994**	.9298	**1.000**	**.9636**	**.9997**	.9861	.9342	**.9595**	**.9995**	**.9338**	**.9407**	**.9373**

TABLE III
EFFECTIVENESS EVALUATION OF THE EXCHANGE DATASET UNDER DIFFERENT GRANULARITY

Exchange	Service				Path			
	AUC	*Precision*	*Recall*	$F1_{best}$	*AUC*	*Precision*	*Recall*	$F1_{best}$
Tprof	/	.3750	.1666	.2307	/	.1750	.0443	.0707
TraceAnomaly	.5105	.5000	.0222	.0426	.5037	.0667	.0127	.0213
MultiModalTrace	.8832	.2672	.5906	.3679	.4031	.1441	.4755	.2211
AutoEncoder	.8807	.4699	.4561	.4629	.9069	.3505	.4510	.3945
DeepSVDD	.8849	.3786	.4561	.4138	.8850	.2135	.5070	.3005
ECOD	.7968	.2405	.4094	.3030	.8556	.1707	.4476	.2471
IForest	.7958	.1614	.6257	.2566	.8317	.1183	.5350	.1938
KNN	.9157	.3305	**.6842**	.4457	.8964	.1766	**.6923**	.2814
LUNAR	.8523	.3788	.4386	.4065	.8174	.1460	.5105	.2271
SUOD	.7975	.1946	.5029	.2806	.8558	.1550	.4580	.2317
TraceArk	**.9713**	**.6707**	.6433	**.6567**	**.9680**	**.5733**	.6154	**.5936**

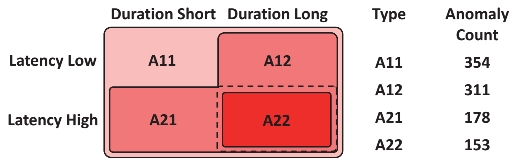
component, which means that the assessment is customized. Moreover,*TraceArk* is lightweight and efficient to apply when compared with the neural network model of *MultiModalTrace*.

Also, *TraceArk* has a 20.34% improvement over the best monitoring-based approach *LUNAR* (i.e., 0.9373 of *TraceArk* v.s. 0.7789 of *LUNAR*). Similar to *MultiModalTrace*, *LUNAR* achieves high *Recall* while having more false alarm samples, i.e., low *Precision*. This result shows that those methods detecting anomalies of single time-series data point would ignore other features of anomalies in practice(i.e. continuity), and thus is more sensitive to temporary anomalies (i.e., spikes in the time-series data) and produces more false alarms.

Table III provides the effectiveness evaluation results of *TraceArk* and other SOTA baselines on the Exchange dataset. The results demonstrate that *TraceArk* outperforms other baselines and achieves the best *F1* results on Exchange. Specifically, *TraceArk* achieves a 168.48% improvement over the best trace-based anomaly alerting approach (i.e., 0.5936 of *TraceArk* v.s. 0.2211 of *MultiModalTrace*) and provides a 50.47% improvement over the best monitoring-based approach (i.e., 0.5936 of *TraceArk* v.s. 0.3945 of *AutoEncoder*). Compared with the TrainTicket dataset, *TraceArk* shows a more significant advantage over the other approaches on the more complex Exchange dataset, indicating that *TraceArk* is more robust on complex data.

C. Feedback Evaluation

We then evaluate feedback mechanism on the two datasets using different feedback strategies. The experiment not only evaluates the effectiveness of feedback in improving the alerting accuracy, but also shows the ability of adapting to different anomaly scenarios and engineers' preferences.



Type	Anomaly Count
A11	354
A12	311
A21	178
A22	153

Fig. 9. TrainTicket anomaly division

We first run *TraceArk* to obtain the anomaly scores and other anomaly-related features, e.g., *NormalizedCount*, *MaxLatency*, *Overhead*, etc., of all alerting subjects in the dataset discussed in Section III-B. We denote the above results as the intermediate data and then split the samples of the intermediate data randomly and equally into a training dataset and a testing dataset, e.g., randomly split the 11210 samples in TrainTicket (Path) dataset into two datasets of 5605 samples. In each round of experiments, we take a limited number of samples (e.g., ten samples each round) from the training dataset as feedback to tune the XGBoost model and then verify the performance of the tuned anomaly alerting model on the testing dataset. We repeat the above steps until the end of the experiment. Moreover, we perform the feedback experiment (including dataset division) 100 times and report the average value and error range of multiple experiments.

Meanwhile, to demonstrate the adaptability of our feedback module under different projects and engineers' preferences for actionable anomalies, we divide the anomalies into four types based on their severity. To avoid data bias that might be introduced by manual division, we divide the anomalies into four categories based on the median latency and duration of the original 354 anomalies. Specifically, as shows in Figure 9, we first divide the anomalies into Latency Low and Latency High categories based on the median latency and divide the anomalies into Duration Short and Duration Long based on the median duration. Finally, four types of anomaly labels are introduced according to the division in Figure 9, where anomaly A11 contains all 354 anomalies, and anomaly A22 includes only the 153 anomalies in both Latency High and

Duration Long categories. The four anomaly types introduce more complexity to actionable anomaly alerting solution and provide good settings for evaluating the adaptability of the feedback module to different actionable anomaly preferences.

1) Feedback Strategy: Feedback strategy is designed to choose what samples should be recommended to engineers for labeling among all the detected subjects. Following previous studies [18], we selected three popular feedback strategies in this field (i.e., ***Top***, ***Spread*** and ***Around*** strategy). We proposed a new strategy ***Kmeanpp*** that selects the n centers determined by the kmean++ algorithm as the recommended samples, i.e., the n samples with significant variances. In addition, we also add a previous feedback approach as a baseline for our feedback module, which uses isolation forest and active learning to adapt the model to the engineer's feedback, so we call it ***IF-Active***.

Subsequently, we evaluated the effectiveness of the four strategies on the Exchange and TrainTicket datasets. As shown in Figure 10 and Figure 11, the *Kmeanpp* strategy performs better in the initial stage. In particular, *Kmeanpp* can achieve the best *F1* score with only 10 to 30 feedback samples in all datasets. Meanwhile, *Kmeanpp* also can achieve comparable performance to *Top* in the late stage. For example, *Top* can achieve the best performance after getting 100 feedbacks in TrainTicket A11 and also can achieve a slightly better performance than *Kmeanpp* in the Exchange dataset after 40 feedback samples. However, in the late stage, the *Kmeanpp* strategy achieve the best *F1* score at the other three dataset (i.e., TrainTicket A12, A21, A22). For example, *Kmeanpp* slightly outperforms *Top* at TrainTicket A12 dataset after 130 feedback samples and outperforms *Top* at TrainTicket A22 dataset after 150 feedback samples. Moreover, the *Kmeanpp* strategy substantially outperforms previous baseline *IF-Active* at Exchange and TrainTicket A11 A21 A22 datasets. This implies that *Kmeanpp* strategy can recommend more diverse samples than other methods, and thus achieve the best performance in the initial stage. While in late stage, the *Top* strategy covers a sufficiently diverse sample set and eventually performs as well as *Kmeanpp* strategy.

As shown in Figure 10, *TraceArk* can achieve an *F1* score of 0.74 with the the feedback mechanism, comparing to at most 0.5936 *F1* score without feedback shown in Table III. We can conclude that the feedback module delivers significant performance improvement on the Exchange dataset.

2) Recommendation Count: To investigate the effect of different recommendation sample numbers per round (i.e., n in the feedback strategy) on *TraceArk* performance, we evaluate the effectiveness of different n under two datasets with *Kmeanpp*. As the evaluation results in Figure 12 demonstrate, *TraceArk* performs poorly in the initial stage when n is small. However, when the number of labeled samples increases, the *F1* score tends to be consistent for different recommendation numbers. As shown in Figure 12a, the *F1* score converges for different n when the number of labeled samples exceeds 100. Based on this finding, we should recommend more unlabeled samples (such as n=20 or n=30) to engineers at the early stage of *TraceArk* deployment and reduce the number of recommendations per round after the number of feedback samples becomes larger to reduce the labeling effort of engineers.

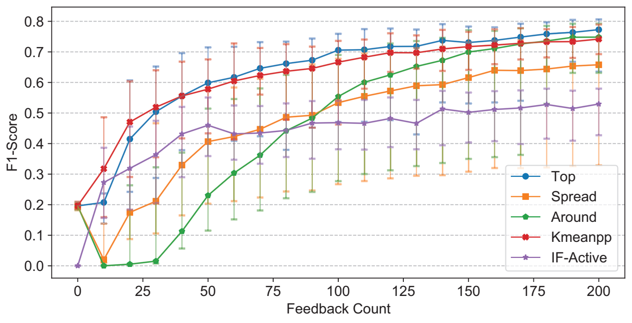


Fig. 10. Exchange feedback evaluation result. Where the x-axis represents the number of accumulated feedback samples and the y-axis represents the corresponding *F1* of *TraceArk* after fine-tuning.

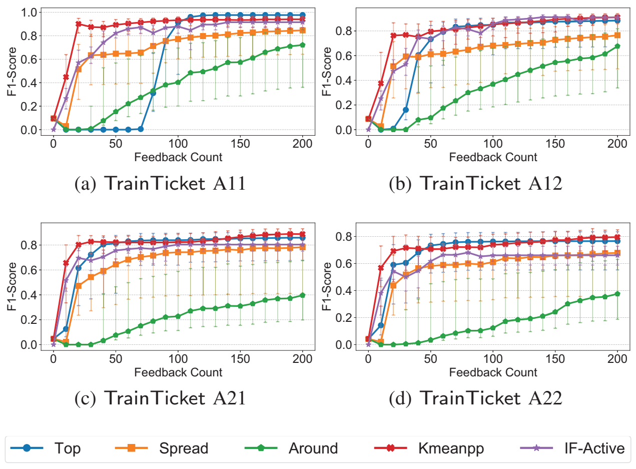


Fig. 11. TrainTicket feedback result

V. Production Impact and Discussion

A. How effective is TraceArk in real scenario?

TraceArk has been transferred to the Exchange team and is applied to the daily anomaly alerting of the Exchange service, which contains several hundred microservices spread over O(100) datacenters worldwide and can generate O(1B) new trace data every day. Currently, *TraceArk* has been running

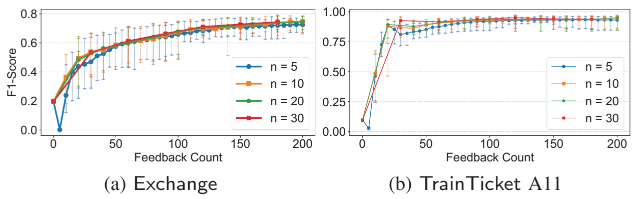
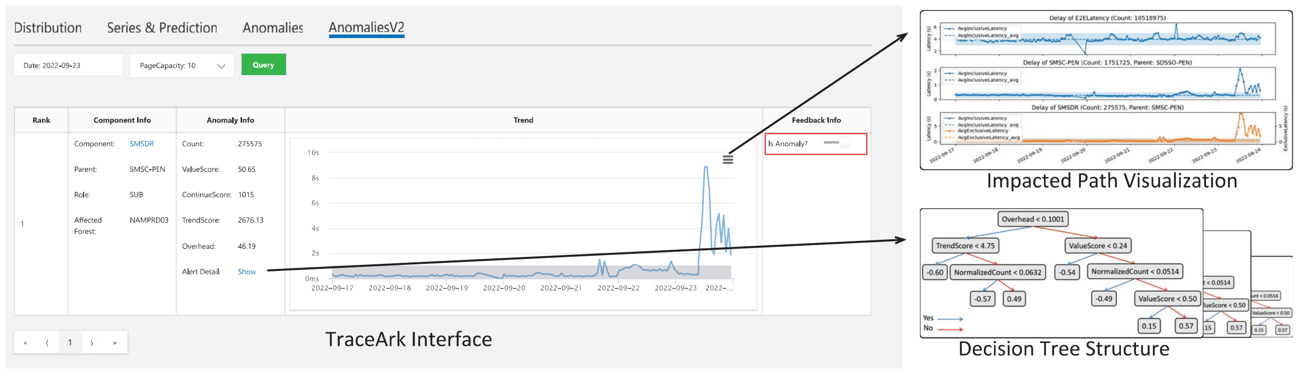


Fig. 12. Recommendation count experiment



Date: 2022-09-23 PageCapacity: 10

Rank	Component Info	Anomaly Info	Feedback Info
1	Component: SMSDR Parent: SMSC-PEN Role: SUB Affected Forest: NAMPRD03	Count: 275575 ValueScore: 50.65 ContinueScore: 1015 TrendScore: 2676.13 Overhead: 46.19 Alert Detail: Show	Is Anomaly?

Fig. 13. *TraceArk* interface of Exchange service. TraceArk Interface is the home page of *TraceArk*. Impacted Path Visualization demonstrates the end to end latency of Exchange, the inclusive latency of parent component and the latency of alert component from top to bottom. Decision Tree Structure demonstrates the structure of the decision tree in *TraceArk*.

steadily online for four months (from June to October). In the last few dozens of days, *TraceArk* reported 118 actionable anomalies to engineers, and 107 of them were confirmed by engineers, achieving a *Precision* of 0.9068. The previous approach adopted by the Exchange team is based on statistical indicators which only has a 0.38 *Precision* in practice. Therefore, *TraceArk* provides a 2.38X improvement in *Precision* and dramatically reduces the false alarm rate and eases the workload of engineers. Moreover, engineers express that the model details and the impact path of the alerted component *TraceArk* help reduce their fault diagnosis time. The runtime overhead is about 2690.59 seconds in average, which satisfies the requirement (i.e., 1 hour) for the engineer to complete anomaly alerting for the online service.

B. How interpretable are the TraceArk and its results?

To justify the interpretability of *TraceArk*, we further visualize the deployed *TraceArk* interface of the Exchange system in Figure 13. In particular, "TraceArk Interface" is the home page for engineers to review the alert results. On this page, engineers can query the alert results for each day, and each result contains information including Rank, Component Info, Anomaly Info, Trend, and Feedback Info. The "Rank" is the severity ranking of the component in the daily alert results (based on the XGBoost prediction). "Component Info" includes basic information about the component, such as component name, parent component name, and the data center it belongs to. "Anomaly Info" contains the features of anomaly evaluation, i.e., three anomaly scores and other features. In addition, in the "Alert Detail" item, engineers can further explore the decision logic of *TraceArk*. Note that to construct an understandable anomaly alerting algorithm, we set the number of sub-decision trees of the XGBoost model to 3 in practice. Whereas previous studies tended to integrate hundreds of decision trees in their algorithms, it is difficult to interpret such algorithms. The "Decision Tree Structure" figure shows one of the decision tree structures of *TraceArk*, and such a decision tree has a depth of 5 and 6 inter-nodes which is acceptable to engineers. The "Trend" column shows the exclusive latency time series of the alert component. In addition, in the upper right menu, engineers can further view other time-series indicators (e.g., 95th exclusive latency) and the impacted path of the component (i.e., "Impacted Path Visualization" on the right) to facilitate further diagnosis of the anomaly. Finally, in the Feedback Info column, we provide an on-the-fly feedback mechanism that allows engineers to provide quick feedback on anomalies when checking the alert results of the day to reduce the labeling effort of engineers.

VI. Related Work

Anomaly alerting in online service systems is a promising research domain, many researchers have investigated the topic from different perspectives and proposed their solutions.

Log-based anomaly alerting techniques [24], [40]–[42] perform data mining on the log data generated by the service system to identify the data patterns of normal logs, while logs that do not match the normal patterns are flagged as anomalies. For example, LogAD [40] integrates multiple anomaly detection approaches and domain knowledge to tackle diverse abnormal log patterns. Li et al. [41] employ several natural language processing techniques to improve the effectiveness of log-based anomaly alerting approach.

Distributed tracing-based anomaly alerting techniques [13], [15], [43], [44] model the pattern of normal traces by analyzing trace structure and component delays and applying this model for detecting anomalies in the system. For example, TraceAnomaly [15] trains a deep bayesian neural network on normal trace data and then uses such network to compute the log-likelihood of a newly generated trace being present in normal data, and finally treats the samples with low log-likelihood as anomalies. MultiModalTrace [13] trains a multi-modal LSTM neural network on normal data. Alertrank [43] adopts XGBoost ranking algorithm to identify the severe alerts out of all incoming alerts based on a set of powerful features (e.g., textual and temporal alert features).

Monitoring-based anomaly alerting techniques address anomaly alerting in a much simpler way. Instead of processing logs or trace data generated by the system, they install a monitor for each server to check the status of the service directly and set specific SLOs for each monitor, such that an anomaly arises when the SLOs are exceeded [7]–[9]. MicroRCA [45] represents the KPIs of the monitor at each time point as a vector, and then perform anomaly alerting. Besides, several studies have proposed that it is possible to introduce a few human feedbacks to improve alerting accuracy. For example, IF-Active [17] and iRRCF-Active [18] improve the performance of baseline models by using human feedback samples and active learning.

In a nutshell, log-based and trace-based anomaly alerting techniques usually package the anomaly alerting of different service components into one large system anomaly alerting task, i.e., they need to use one model to learn the data patterns of different service components simultaneously. As data patterns are highly diverse from service to service, meaning that this is a challenging task, resulting in poor performance for this type of approach. Meanwhile, the deployment overhead of monitor-based techniques can be high, as new monitors need to be deployed within the system to check the status of the service directly. Moreover, the above methods rarely have feedback mechanisms to accommodate different project preferences for anomaly thresholds. Lastly, their interpretability is insufficient, e.g., the interpretability of the textual features adopted by log-based techniques and Alertrank has been much criticized, which prevents engineers from working with them with confidence. Yet the *TraceArk* proposed in this paper solves the above problems.

VII. Conclusion

In this paper, we propose an actionable performance anomaly alerting approach *TraceArk* based on trace data for online service systems. It contains an anomaly assessment method that takes into account both temporal latency information, trace structural characteristics and the engineers' perspective on the anomaly. Meanwhile, *TraceArk* also incorporates a small amount of engineer experience (i.e., feedback) using a simple structured tree-based model to learn actionable anomaly thresholds, and could adapt in different anomalous scenarios. Our comprehensive experiments show that *TraceArk* significantly outperforms the existing state-of-the-art approaches. Furthermore, *TraceArk* has been deployed and running stably for four months in a real production environment with high anomaly alerting accuracy. The interpretable alerting details help engineers understand the alerting process and guide further diagnosis.

References

[1] C. Bansal, S. Renganathan, A. Asudani, O. Midy, and M. Janakiraman, "Decaf: Diagnosing and triaging performance issues in large-scale cloud services," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, 2020, pp. 201–210.

[2] M. Ma, Y. Liu, Y. Tong, H. Li, P. Zhao, Y. Xu, H. Zhang, S. He, L. Wang, Y. Dang *et al.*, "An empirical investigation of missing data handling in cloud node failure prediction," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1453–1464.

[3] Y. Liu, H. Yang, P. Zhao, M. Ma, C. Wen, H. Zhang, C. Luo, Q. Lin, C. Yi, J. Wang *et al.*, "Multi-task hierarchical classification for disk failure prediction in online service systems," in *Proceedings of the SIGKDD Conference on Knowledge Discovery and Data Mining (SIGKDD)*. ACM, 2022, pp. 3438–3446.

[4] Y. Liu, X. Zhang, S. He, H. Zhang, L. Li, Y. Kang, Y. Xu, M. Ma, Q. Lin, Y. Dang, S. Rajmohan, and D. Zhang, "Uniparser: A unified log parser for heterogeneous log data," in *Proceedings of the Web Conference*. ACM, 2022, p. 1893–1901.

[5] T. Zheng, X. Zheng, Y. Zhang, Y. Deng, E. Dong, R. Zhang, and X. Liu, "Smartvm: a sla-aware microservice deployment framework," *World Wide Web*, vol. 22, no. 1, pp. 275–293, 2019.

[6] F. Ramírez, C. Mera-Gómez, R. Bahsoon, and Y. Zhang, "An empirical study on microservice software development," in *2021 IEEE/ACM Joint 9th International Workshop on Software Engineering for Systems-of-Systems and 15th Workshop on Distributed Software Development, Software Ecosystems and Systems-of-Systems (SESoS/WDES)*. IEEE, 2021, pp. 16–23.

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

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

[9] M. Ma, S. Zhang, J. Chen, J. Xu, H. Li, Y. Lin, X. Nie, B. Zhou, Y. Wang, and D. Pei, "Jump-starting multivariate time series anomaly detection for online service systems," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 413–426.

[10] X. Guo, X. Peng, H. Wang, W. Li, H. Jiang, D. Ding, T. Xie, and L. Su, "Graph-based trace analysis for microservice architecture understanding and problem diagnosis," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1387–1397.

[11] L. Huang and T. Zhu, "tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces," in *Proceedings of the ACM SoCC*, 2021, pp. 76–91.

[12] "Jaeger," Jaeger, 2022, https://www.jaegertracing.io/.

[13] S. Nedelkoski, J. Cardoso, and O. Kao, "Anomaly detection from system tracing data using multimodal deep learning," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 179–186.

[14] J. Chen, S. Zhang, X. He, Q. Lin, H. Zhang, D. Hao, Y. Kang, F. Gao, Z. Xu, Y. Dang *et al.*, "How incidental are the incidents? characterizing and prioritizing incidents for large-scale online service systems," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 373–384.

[15] P. Liu, H. Xu, Q. Ouyang, R. Jiao, Z. Chen, S. Zhang, J. Yang, L. Mo, J. Zeng, W. Xue *et al.*, "Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2020, pp. 48–58.

[16] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu, "Characterizing microservice dependency and performance: Alibaba trace analysis," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 412–426.

[17] M. A. Siddiqui, A. Fern, T. G. Dietterich, R. Wright, A. Theriault, and D. W. Archer, "Feedback-guided anomaly discovery via online optimization," in *Proceedings of the 24th ACM SIGKDD*, 2018, pp. 2200–2209.

[18] Y. Wang, Z. Wang, Z. Xie, N. Zhao, J. Chen, W. Zhang, K. Sui, and D. Pei, "Practical and white-box anomaly detection through unsupervised and active learning," in *29th international conference on computer communications and networks*. IEEE, 2020, pp. 1–9.

[19] A. S. Acharya, A. Prakash, P. Saxena, and A. Nigam, "Sampling: Why and how of it," *Indian Journal of Medical Specialties*, vol. 4, no. 2, pp. 330–333, 2013.

[20] Q. Lin, J.-G. Lou, H. Zhang, and D. Zhang, "idice: problem identification for emerging issues," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 214–224.

[21] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[22] D. Yarowsky, “Unsupervised word sense disambiguation rivaling supervised methods,” in *33rd annual meeting of the association for computational linguistics*, 1995, pp. 189–196.

[23] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao, “Benchmarking microservice systems for software engineering research,” in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 323–324. [Online]. Available: https://doi.org/10.1145/3183440.3194991

[24] C. Zhang, X. Peng, C. Sha, K. Zhang, Z. Fu, X. Wu, Q. Lin, and D. Zhang, “Deeptralog: Trace-log combined microservice anomaly detection through graph-based deep learning,” 2022.

[25] Z. Li, N. Zhao, M. Li, X. Lu, L. Wang, D. Chang, X. Nie, L. Cao, W. Zhang, K. Sui *et al.*, “Actionable and interpretable fault localization for recurring failures in online service systems,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 996–1008.

[26] “Kubernetes,” Kubernetes, 2022, https://kubernetes.io/.

[27] “chaos-mesh,” chaos-mesh, 2022, https://chaos-mesh.org/.

[28] J. Soldani and A. Brogi, “Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey,” *ACM Computing Surveys (CSUR)*, vol. 55, no. 3, pp. 1–39, 2022.

[29] C. C. Aggarwal, “An introduction to outlier analysis,” in *Outlier analysis*. Springer, 2017, pp. 1–34.

[30] L. Ruff, R. Vandermeulen, N. Goernitz, L. Deecke, S. A. Siddiqui, A. Binder, E. Müller, and M. Kloft, “Deep one-class classification,” in *International conference on machine learning*. PMLR, 2018, pp. 4393–4402.

[31] Z. Li, Y. Zhao, X. Hu, N. Botta, C. Ionescu, and G. Chen, “Ecod: Unsupervised outlier detection using empirical cumulative distribution functions,” *IEEE Transactions on Knowledge and Data Engineering*, 2022.

[32] F. T. Liu, K. M. Ting, and Z.-H. Zhou, “Isolation forest,” in *2008 eighth ieee international conference on data mining*. IEEE, 2008, pp. 413–422.

[33] S. Ramaswamy, R. Rastogi, and K. Shim, “Efficient algorithms for mining outliers from large data sets,” in *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, 2000, pp. 427–438.

[34] A. Goodge, B. Hooi, S.-K. Ng, and W. S. Ng, “Lunar: Unifying local outlier detection methods via graph neural networks,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, no. 6, 2022, pp. 6737–6745.

[35] Y. Zhao, X. Hu, C. Cheng, C. Wang, C. Wan, W. Wang, J. Yang, H. Bai, Z. Li, C. Xiao *et al.*, “Suod: Accelerating large-scale unsupervised heterogeneous outlier detection,” *Proceedings of Machine Learning and Systems*, vol. 3, pp. 463–478, 2021.

[36] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li *et al.*, “Robust log-based anomaly detection on unstable log data,” in *Proceedings of the 27th ESEC/FSE Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019, pp. 807–817.

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

[38] A. Bernieri, G. Betta, and C. Liguori, “On-line fault detection and diagnosis obtained by implementing neural algorithms on a digital signal processor,” *IEEE Transactions on Instrumentation and Measurement*, vol. 45, no. 5, pp. 894–899, 1996.

[39] Y. Zhao, Z. Nasrullah, and Z. Li, “Pyod: A python toolbox for scalable outlier detection,” *Journal of Machine Learning Research*, vol. 20, no. 96, pp. 1–7, 2019.

[40] N. Zhao, H. Wang, Z. Li, X. Peng, G. Wang, Z. Pan, Y. Wu, Z. Feng, X. Wen, W. Zhang *et al.*, “An empirical investigation of practical log anomaly detection for online service systems,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1404–1415.

[41] Z. Li, J. Zhang, X. Zhang, F. Lin, C. Wang, and X. Cai, “Natural language processing-based model for log anomaly detection,” in *2022 IEEE 2nd International Conference on Software Engineering and Artificial Intelligence (SEAI)*. IEEE, 2022, pp. 129–134.

[42] M. Du, F. Li, G. Zheng, and V. Srikumar, “Deeplog: Anomaly detection and diagnosis from system logs through deep learning,” in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 1285–1298.

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

[44] C. Zhang, T. Zhou, Q. Wen, and L. Sun, “Tfad: A decomposition time series anomaly detection architecture with time-frequency analysis,” in *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, 2022, pp. 2497–2507.

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