MicroSketch: Lightweight and Adaptive Sketch Based Performance Issue Detection and Localization in Microservice Systems

Yufeng Li[1], Guangba Yu[2], Pengfei Chen[2](✉), Chuanfu Zhang[1], and Zibin Zheng[2]

[1] School of Systems Science and Engineering, Sun Yat-Sen University, Guangzhou, China
liyf323@mail2.sysu.edu.cn, zhangchf9@mail.sysu.edu.cn

[2] School of Computer Science and Engineering, Sun Yat-Sen University, Guangzhou, China
yugb5@mail2.sysu.edu.cn, {chenpf7,zhzibin}@mail.sysu.edu.cn

Abstract. With the rapid growth of microservice systems in cloud-native environments, end-to-end traces have become essential data to help diagnose performance issues. However, existing trace-based anomaly detection and root cause analysis (RCA) still suffer from practical issues due to either the massive volume or frequent system changes. In this study, we propose a lightweight and adaptive trace-based anomaly detection and RCA approach, named *MicroSketch*, which leverages Sketch based features and Robust Random Cut Forest (RRCForest) to render trace analysis more effective and efficient. In addition, *MicroSketch* is an unsupervised approach that is able to adapt to changes in microservice systems without any human intervention. We evaluated *MicroSketch* on a widely-used open-source system and a production system. The results demonstrate the efficiency and effectiveness of *MicroSketch*. *MicroSketch* significantly outperforms start-of-the-art approaches, with an average of 40.9% improvement in F1 score on anomaly detection and 25.0% improvement in Recall of Top-1 on RCA. In particular, *MicroSketch* is at least 60x faster than other methods in terms of diagnosis time.

Keywords: Microservice · Anomaly detection · Root cause analysis · Sketch

1 Introduction

Over the years, more and more enterprises (e.g., Amazon, Netflix, and Twitter) have gradually replaced monolithic applications with loosely-coupled and lightweight microservices [2,16]. The loosely-coupled paradigm of microservice applications enables independent refactoring and dynamic scaling for each service [19,20]. Despite various resilience strategies in modern microservice architecture (e.g., load balancing and circuit breaking), system-wide issues of microservice

J. Troya et al. (Eds.): ICSOC 2022, LNCS 13740, pp. 219–236, 2022.
https://doi.org/10.1007/978-3-031-20984-0_15

applications are still pervasive due to resource exhaustion, network jam, etc. Performance issues that manifest themselves as high latency are easier to happen but more difficult to diagnose than availability issues [4].

Distributed tracing [14] becomes a mainstream tool for troubleshooting in microservice systems. Distributed tracing records the detailed executions of completing a user request, including the invocation paths of service instances and latency information of these invocations between service instances. Because distributed tracing has an irreplaceable advantage in capturing interactions between service instances, it is becoming an indispensable infrastructure for monitoring, profiling, analyzing and diagnosing in modern distributed software systems, especially in large microservice applications. However, current tracing tools (e.g., Jaeger[1] and Zipkin[2] are primarily designed to collect and present traces rather than automatically diagnose performance issues.

It is an error-prone and labor-intensive process to manually detect performance issues and localize root causes based on current tracing tools. Therefore, some automated trace analysis approaches have been proposed in microservice systems [6,10,18]. However, state-of-the-art studies with traces for performance analysis encounter practical issues due to the massive volume of traces or frequent system changes. As shown in Table 1, tprof [6] takes over 600 s and MicroRank [18] needs over 100 s to infer root causes by analyzing 10,000 traces when one fault occurs. This is because tprof [6] hierarchically groups traces by request types and trace structures, and calculates increasingly detailed aggregated statistics, which consumes a great deal of time. MicroRank introduces PageRank to calculate the weights of traces, which needs a long time to get the converged results when meeting a larger scale of traces. The inference time will be further exacerbated when a larger-scale microservice system is encountered. TraceAnomaly [10] takes less time to infer root causes than MicroRank, but it needs to retrain the deep Bayesian network after microservice updates. In addition, this training process is extremely time-consuming, resulting in poor adaptability.

Table 1. Resource overhead and inference time for some state-of-the-art trace analysis systems. (The experiment platform is shown in Subsect. 4.1)

System	Method	CPU utilization (%)	Memory usage (MB)	Time(s)
tprof [6]	Hierarchical analysis	12 ± 2 single core	800 ± 50	600 ± 30
TraceAnomaly [10]	Deep Bayesian network	75 ± 5 single core	550 ± 50	65 ± 10
MicroRank [18]	PageRank+Spectrum	12 ± 2 single core	430 ± 50	105 ± 10

To address the above drawbacks of existing work, we propose *MicroSketch*, which leverages Sketch [11] based features and Robust Random Cut Forest (RRCForest) [5] to detect performance issues and localize root causes using distributed traces in a lightweight and adaptive way, with a low time and space com-

[1] Jaeger, https://jaegertracing.io/.

[2] Zipkin, https://zipkin.io/.

plexity. It consists of three main procedures including *Status Encoder*, *Anomaly Detector*, and *Fault Locator*. *Status Encoder* collects trace data and encodes these data into a status vector in order to conduct *Anomaly Detector*. Then *Anomaly Detector* determines whether it is an anomaly. Once an anomaly is detected, *Fault Locator* is triggered and generates a ranking list containing possible root causes for the anomaly. We evaluated *MicroSketch* on a widely-used open-source system and a production system. The results demonstrate the efficiency and effectiveness of *MicroSketch*. Moreover, *MicroSketch* significantly outperforms start-of-the-art approaches, with an average of 40.9% improvement in F1 score on anomaly detection and 25.0% improvement in Recall of Top-1 on root cause analysis (RCA). In particular, *MicroSketch* is at least 60x faster than other methods in terms of diagnosis time. Besides, *MicroSketch* has the ability to automatically adapt to the changes of microservice systems and continually work without any manual intervention.

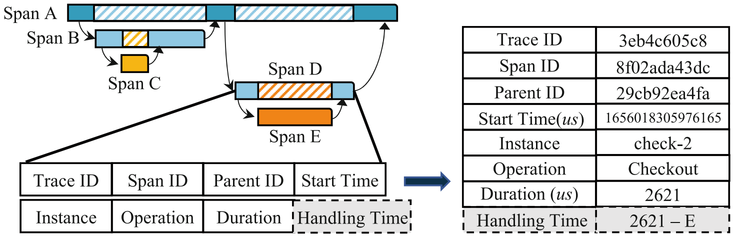
Overall, the contribution of this paper is three-fold summarized as follows.

- We improve the DDSketch, state-of-the-art sketch technology, so that it keeps all the original features while reducing storage space to calculate the quantiles with sublinear space and linear time complexity.
- We propose a novel anomaly detection and RCA approach in microservice environments based on the adaptive RRCForest, which automatically adapts to variable-length input vector and renders our model appropriate for dynamic microservice systems.
- We implement *MicroSketch* to detect performance issues and localize root causes in a lightweight and adaptive way. We conduct extensive experiments based on a widely-used microservice benchmark and a production microservice system. Experimental results demonstrate that *MicroSketch* achieves good results both on anomaly detection and RCA. In addition, *MicroSketch* is at least 60x faster than other methods in terms of diagnosis time.

2 Background

Distributed tracing is an important technique for gaining insight and observability into microservice systems [15]. In large-scale microservice systems, a request is typically handled by multiple services deployed in different nodes or even data centers. Distributed tracing provides a method to track the complete execution path of each request. A span represents a logical unit of execution, handled by an operation of a service instance in a microservice system. All spans that serve for the same request collectively form a trace, as illustrated in the left part of Fig. 1. Spans generated by the same request have the same trace ID. For each span, it records some attributes (i.e., Trace ID, Span ID, and Start time), as shown on the right part of Fig. 1.

As shown in Fig. 1, the duration of a span is the accumulated time spent by this operation and all downstream operations. Therefore, when the duration of span E increases due to a fault, all upstream spans of E (i.e., span A and D) will increase as well due to fault propagation, making it difficult to determine which span is the root cause. To overcome this problem, we transform duration into a more directional metric. For each span, we subtract the duration of its all child spans from its duration to get its real handling time. In Fig. 1, the non-shaded part is called the span's handling time.



Trace ID	3eb4c605c8
Span ID	8f02ada43dc
Parent ID	29cb92ea4fa
Start Time(*us*)	1656018305976165
Instance	check-2
Operation	Checkout
Duration (*us*)	2621
Handling Time	2621 – E

Fig. 1. An example of trace with five spans in Hipster-Shop (Hipster-Shop, https://github.com/GoogleCloudPlatform/microservices-demo).

3 System Design

3.1 System Overview

Figure 2 demonstrates the framework of *MicroSketch*. It consists of three modules, including *Status Encoder*, *Anomaly Detector* and *Fault Locator*. We use *time interval* to denote the trace analysis frequency (1 min default in this study). Firstly, given the traces in a time interval, *Status Encoder* leverages the extended DDSketch to calculate the quantile of the handling time for each invocation group and encodes them as status vector $\boldsymbol{x} = (x_1, x_2, ..., x_m)$ (Subsect. 3.2). Secondly, *Anomaly Detector* analyzes the status vector based on adaptive Robust Random Cut Forest (RRCForest) and outputs the anomaly score of $\boldsymbol{x}$ (Subsect. 3.3). If the score of $\boldsymbol{x}$ is over the predefined threshold τ, *Fault Locator* is triggered to determine the root cause (Subsect. 3.4).

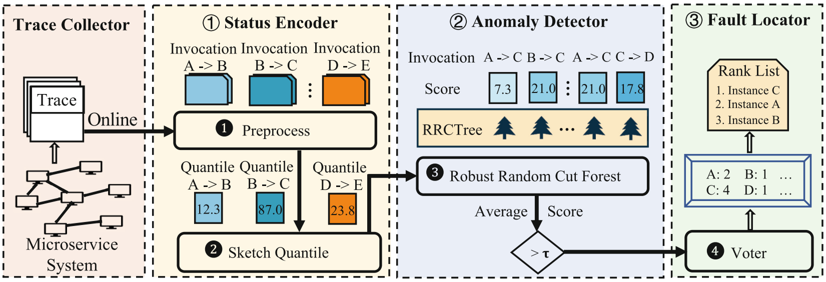


Fig. 2. The framework of *MicroSketch*

3.2 Status Encoder

At each time interval, *MicroSketch* queries all traces in the time interval and inputs them into *Status Encoder.*

Status Vector. Quantile is a splendid statistic for profiling data, especially for latency data. The quantile of the handling time, such as the 50th or the 90th percentiles, reflects the quality of service instance. As shown in Fig. 3, p90 of the operation (product-1.sql-query) rises when issue occurs. Quantile [11] can be formalized as follows. Given a multiset S of size n, the q-quantile item $x_q \in S$ is the item x whose index $R(x)$ in sorted multiset S is $\lfloor 1 + q(n-1) \rfloor$ for $0 \leq q \leq 1$.

After all spans are collected, we group them by the type of invocation. The invocation owning the same upstream service instance, same downstream service instance and same operation belongs to the same type. We calculate the quantile (p90 in this paper) of the handling time for each group and manage these quantiles as a vector $\boldsymbol{x} = [x_1, x_2, ..., x_m]$, where x_i means the quantile of the handling time that belongs to the invocation group i. Status vector $\boldsymbol{x}$ largely reflects the performance status of the global microservice system in this time interval.

Sketch Technology. Commonly, we sort the multi-set S first and then query by index $\lfloor 1 + q(n-1) \rfloor$ to generate an exact q-quantile, but it requires huge computing resources and time for sort. Nevertheless, it is not so necessary to get the exact quantile value in our scenarios. An estimated quantile that does not deviate too far from the exact value can also be enough to conduct anomaly detection. Therefore, we introduce Distributed Distribution Sketch (DDSketch) [11], which is able to calculate the quantile much faster and more economically with relative-error guarantees and sublinear space and linear time complexity. DDSketch keeps rigorously relative-error guarantees by dividing the data stream into fixed buckets. It means that, given a parameter α, each estimated q-quantile $\tilde{x}_q$ and the exact q-quantile x_q are satisfied to $|\tilde{x}_q - x_q| \leq \alpha x_q$.

However, we do not need to satisfy the rigorous relative-error guarantees. Because we hardly focus on the head latency data (i.e., the main-body distribution of latency data). We use an equi-width histogram to extend the DDSketch, which allows us to reduce memory usage without losing the relative-error guarantees in the tail data, compared to DDSketch. To elaborate on how extended DDSketch works, the three phases, namely *initialization*, *insertion* and *query* are summarized.

In the phase of initialization, we define the *tail relative-error rate* α, *boundary* L and *head granularity factor* β to keep the error guarantees. Given a quantile percentage q, if $x_q < L$, the estimated quantile $\tilde{x}_q$ will be satisfied to $|\tilde{x}_q - x_q| \leq \beta$ and if $x_q \geq L$, the estimated quantile $\tilde{x}_q$ will be satisfied to $|\tilde{x}_q - x_q| \leq \alpha x_q$. Thus, we keep the relative-error guarantees in the *tail data* (the numbers are greater than L), and reduce the memory usage at the cost of losing the relative-error guarantees in the *head data* (the numbers are less than L). In this paper, α is set as 1%, L is set as p50 estimated by the last or 0 and β is set as $\frac{L}{100}$ or other reasonable values.

Algorithm 1. extended DDSketch Insert Algorithm

```
Input: the number $x \in \mathbb{R} > 0$
if $x < L$ then
    $i \leftarrow \left\lceil \frac{x}{\beta} \right\rceil$        //Find bucket index i if x belongs to the head data
    $H_i \leftarrow H_i + 1$  //Bucket H[i] adds 1
else
    $i \leftarrow \lceil \log_\gamma(x) \rceil$ //Find bucket index i if x belongs to the tail data
    $B_i \leftarrow B_i + 1$  //Bucket B[i] adds 1
end if
```

In the phase of insertion, let $\gamma := \frac{(1+\alpha)}{(1-\alpha)}$. If the input number x is less than L, bucket $H[\left\lceil \frac{x}{\beta} \right\rceil]$ adds 1. Otherwise, bucket $B[\lceil \log_\gamma(x) \rceil]$ adds 1. This is shown in Algorithm 1.

In the phase of query, given a quantile percentage q, extended DDSketch try to find the minimum index i which makes $\sum_{j=0}^{i} H_j > q(n-1)$. If we succeed in finding the index i, extended DDSketch returns the estimated quantile $\frac{2i-1}{2}\beta$. Otherwise, extended DDSketch finds the minimum index i which makes $\sum_{j=0} H_j + \sum_{j=0}^{i} B_j > q(n-1)$ and returns the estimated quantile $\frac{2\gamma^i}{\gamma+1}$. The detail is described in Algorithm 2. Finally, we update L to the estimated p50.

Algorithm 2. extended DDSketch Query Algorithm

```
Input: $0 \le q \le 1$
Output: the estimated q-quantile
count $\leftarrow$ 0     i $\leftarrow -1$
while $i < len(H)$ && count $\le q(n-1)$ do
    i $\leftarrow i + 1$
    count $\leftarrow$ count $+H_i$               //Accumulate bucket H[i] in order
end while
if count $> q(n-1)$ then
    return $\frac{2i-1}{2}\beta$                   //The q-quantile falls in bucket H[i]
end if
while count $\le q(n-1)$ do
    $i \leftarrow \min(\{j : B_j > 0 \wedge j > i\})$ //Accumulate non-empty bucket B[j] in order
    count $\leftarrow$ count $+B_i$;
end while
return $\frac{2\gamma^i}{\gamma+1}$                       //The q-quantile falls in bucket B[i]
```

3.3 Anomaly Detector

After encoding, traces in time interval are transformed into status vector $\boldsymbol{x} = [x_1, x_2, ..., x_m]$. Anomaly detection is converted to outlier detection based on the time series of status vector $\boldsymbol{x}$. Robust Random Cut Forest (RRCForest) [5] is a streaming model and follows the mechanism of isolation forest [9]. In detail, the point set is distributed in a multidimensional space $S \subset \mathbb{R}^m$, for each case, RRCForest randomly chooses a dimension and randomly chooses a value in this dimension to cut. This process is called *dimension cut*. After one dimension cut, the whole space is divided into two subspaces. Subsequently, two subspaces are recursively cut in the same way. A point is determined to be isolated if it occupies a subspace exclusively. The scatter chart in Fig. 3 shows an isolated point occupying the shaded left upper corner exclusively.

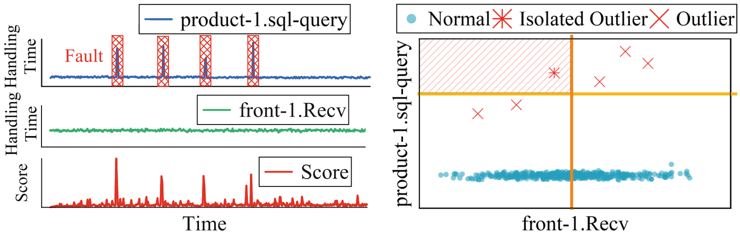


Fig. 3. The distribution of status vectors and the mechanism of RRCForest. The first and second line charts in the left part are p90 handling time of operation front-1.Recv and product-1.sql-query, respectively. The third line chart is the anomaly score given by adaptive RRCForest. The scatter chart is the distribution of status vectors and an example of the dimension cut of a two-dimensional space $S \subset \mathbb{R}^2$.

Taking two operations (front-1.Recv and product-1.sql-query) in HipsterShop (Subsect. 4.1) as an example, we use *Status Encoder* to transform traces in each time interval into status vector $\boldsymbol{x} = [x_1, x_2]$. Both the distribution of each dimension and the distribution of the vectors are shown in Fig. 3. We intermittently injected four anomalies into service instance product-1. There are four peaks in the handling time of product-1 because of fault injection. The scatter chart in Fig. 3 presents that several vectors corresponding to the peaks are labeled as red forks. The red forks can be isolated by two or three dimension cuts and those dense normal blue dots require more cuts to be isolated.

Next, we describe how RRCForest detects anomalies. The process of dimension cut mentioned above is described by a binary tree structure, called Robust Random Cut Tree (RRCTree). As shown in Fig. 4, RRCTree owns two kinds of nodes. One is ***leaf***, represented as a square rectangle, the other is ***branch***, represented as a rounded rectangle. We also summarize three phases for the construction of RRCTree, namely *initialization*, *insertion* and *query*.

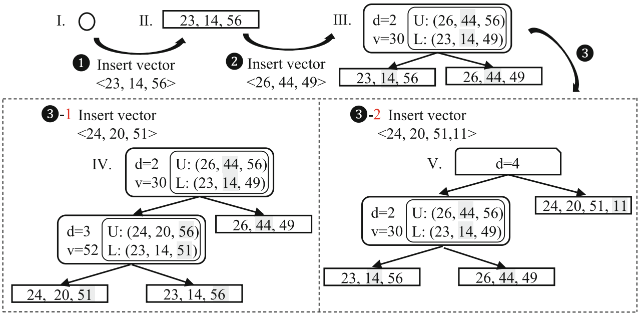


Fig. 4. The construction of RRCTree. U and L keeps the maximum and minimum of each dimension of the leaves to avoid repeatedly calculating them for Eq. 1. d and v denote the cut dimension and cut value, respectively. Each leaf is assigned to a status vector and each branch records how the vectors are isolated.

In the phase of initialization, we create an empty tree, given in Fig. 4-I.

$$w_i = \frac{max_{\boldsymbol{x} \in S} x_i - min_{\boldsymbol{x} \in S} x_i}{\sum_i (max_{\boldsymbol{x} \in S} x_i - min_{\boldsymbol{x} \in S} x_i)}. \tag{1}$$

In the phase of insertion, given a RRCTree T', we insert a vector $\boldsymbol{x}$. Let S' as all vectors in RRCTree T' and $S = S' \cup \boldsymbol{x}$. If RRCTree is empty (case 1), we directly create a leaf, assigned to this vector. Figure 4-❶ shows the case 1.

If RRCTree is non-empty (case 2), we move on to the following discussion. The case 2 is further divided into three sub-cases. We randomly select the *cut dimension* d according to the weight w_i which is calculated in the Eq. 1. After selecting cut dimension d, we randomly and uniformly choose a *cut value* $v \in [min_{\boldsymbol{x} \in S} x_d, max_{\boldsymbol{x} \in S} x_d]$. If $v \leq min_{\boldsymbol{x}' \in S'} x'_d$ (case 2–1), we create a branch and a leaf that is assigned to the vector $\boldsymbol{x}$. Then, we set the created leaf as the left subtree of the created branch and the RRCTree T' as the right subtree. If $v > max_{\boldsymbol{x}' \in S'} x'_d$ (case 2–2), we set the RRCTree T' as the left subtree of the created branch and the created leaf as the right subtree. If neither is the case (case 2–3), we consider inserting the vector $\boldsymbol{x}$ into the left subtree of RRCTree T' or right subtree. In detail, for the cut dimension d' and cut value v' of the root branch, if $x_{d'} \leq v'$, we insert the vector $\boldsymbol{x}$ into the left subtree of RRCTree T'. Otherwise, we insert the vector $\boldsymbol{x}$ into the right subtree. Since the subtree is also a RRCTree, the insertion can run recursively until it goes back to the case 2–1 or case 2–2. In sum, Fig. 4-❷ shows the case 2–1. The case 2–2 is similar to the case 2–1. Figure 4-❸-1 shows the case 2–3.

The last case (case 3) is to insert a variable-length vector $\boldsymbol{x}$ and $len(\boldsymbol{x}) > max_{\boldsymbol{x}' \in S'} len(\boldsymbol{x'})$. We create a branch, named *dimension branch*, which represents that a new dimension occurs. Then, we set RRCTree T' as the left subtree of the created dimension branch and the inserted vector as the right subtree. Figure 4-❸-2 shows the case 3.

Further, in order to prevent the tree from excessively expanding, we set *tree size* (128 in this paper) in advance and delete the earliest point from the tree when the number of leaves exceeds *tree size*. Deletion is similar to insertion.

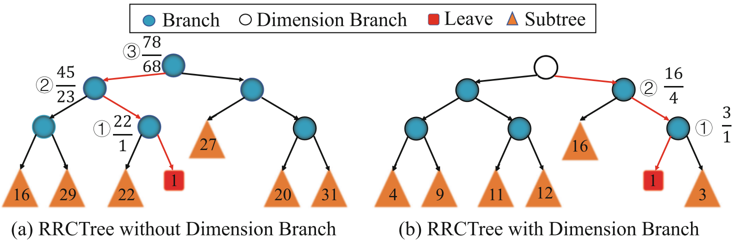
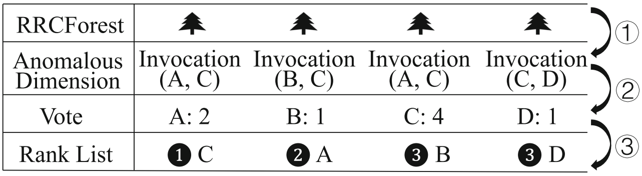


Fig. 5. An example of calculating the anomaly score. (a) The cut rates of red leaf are $\frac{22}{1}$, $\frac{45}{23}$ and $\frac{78}{68}$. The score of the leaf is 22. (b) The cut rates of red leaf are $\frac{3}{1}$ and $\frac{16}{4}$ and the score of the leave is 4. The higher the score, the more anomalous the vector.

In the phase of query, we obtain an anomaly score for the inserted vector. As soon as the insertion of a vector is complete, we query its score. We define the cut rate of a branch as $r = \frac{max(n_{left}, n_{right})}{min(n_{left}, n_{right})}$, where n_{left} is the number of leaves that belong to the left subtree of the branch and n_{right} is the number of leaves that belong to the right subtree of the branch. We find all ancestor branches in the path from the leaf corresponding to the vector to the root branch except dimension branch and calculate these branches' cut rates. Logically, the score of the vector is equal to the maximum in these cut rates. Figure 5 presents two examples of how to calculate the anomaly score of a leaf.

Therefore, anomaly score is closely related to *tree size*. Practically, we give a threshold $\tau = mean \times \log(tree\ size)$, where *mean* denotes the average history score. If one's score exceeds the threshold τ, we regard it as an anomaly.

The above section illustrates how a RRCTree is constructed and queried. As listed in Eq. 1, the selection of cut dimension is random and probabilistic. To make this random construction more convergent to its expectation, we generally build multiple RRCTrees simultaneously and independently. The final score is the average of all RRCTrees' scores. Therefore, in practice, we have to define a parameter *tree number* (50 in this paper by default), which determines how many RRCTrees *Anomaly Detector* maintains.



RRCForest	🌲	🌲	🌲	🌲
Anomalous Dimension	Invocation (A, C)	Invocation (B, C)	Invocation (A, C)	Invocation (C, D)
Vote	A: 2	B: 1	C: 4	D: 1
Rank List	❶ C	❷ A	❸ B	❸ D

Fig. 6. The details of *Fault Locator*. Each RRCTree independently points out an anomalous invocation. The instance C is viewed as the root cause because there are four RRCTrees voting for C.

3.4 Fault Locator

Once *Anomaly Detector* finds an outlier, *Fault Locator* will be triggered. For a RRCTree, the dimension of the branch corresponding to the largest cut rate is considered an anomalous dimension. Each RRCTree points out an anomalous dimension of the outlier. An anomalous dimension represents that one type of invocation is anomalous. If the invocation is anomalous, we conclude that the upstream service instance or downstream service instance may be anomalous. RRCForest gives a set of anomalous invocations $[I(u_1, d_1), I(u_2, d_2), ..., I(u_k, d_k)]$, where u and d denote upstream service instance and downstream service instance, respectively. To further determine the most likely root cause, we propose a voting mechanism. Each anomalous invocation $I(u, d)$ votes for service instance u and service instance d. The service instance with the most votes is regarded as the root cause. Figure 6 presents that instance C is determined as the root cause after the vote of four RRCTrees.

4 Experiment Setup

4.1 Datasets

We use two datasets to validate our approach. One, named $\mathcal{A}$, is based on one of the most widely-used open-source microservice systems, Hipster-Shop. The other, named $\mathcal{B}$, is based on a production microservice system in China Mobile, the largest telecommunication company in China. Table 2 shows some details of our experimental datasets. We implement *MicroSketch* with Python 3.7. All experiments are conducted on a workstation with 4-core 2 GHz Intel Core i5-1038NG7 CPU and 16 GB memory.

Table 2. Experimental datasets

Dataset	Benchmark	Fault number	Fault type	Trace number
$\mathcal{A}$	Hipster-shop	50	Network, CPU, IO read, IO write	2,902 K
$\mathcal{B}$	Production system	8	Network, CPU	168 K

Hipster-Shop Microservice System. This system is an e-commerce website with 10 microservices that are implemented in different programming languages and intercommunicate using gRPC. We continuously run a workload generator, which can simulate real-world users. The microservice benchmark is deployed in a Kubernetes cluster that consists of 1 master node and 5 worker nodes based on virtual machines, which singly run with Ubuntu 18.04 OS. To mimic performance issues, we use two tools, Chaosblade[3] and Strace[4] to inject four types of faults into Hipster-Shop. We injected 50 faults to Hipster-shop in total. Each fault injection lasts for 30 to 60 s.

[3] Chaosblade, https://github.com/chaosblade-io/chaosblade.

[4] Strace, https://strace.io.

Real-World Microservice System. Dataset $\mathcal{B}$, released by the 2020 AIOps Challenge Event, is based on a real-world production microservice system in China Mobile. In particular, the workload of the system in $\mathcal{B}$ is a replica of the real-world workload. The types of faults include network fault and CPU fault. Note that since this event does not only focus on microservice applications, we only selected those faults related to microservices on May 31st, 2020.

4.2 Evaluation Metric

We use **Precision** ($\boldsymbol{P}$), **Recall** ($\boldsymbol{R}$) and **F1 score** ($\boldsymbol{F1}$) to compare the performance of anomaly detection. Precision is computed by $\frac{TP}{TP+FP}$, while Recall is computed by $\frac{TP}{TP+FN}$, where TP, FP and FN refer to the number of anomalous time intervals that are correctly predicted to be anomalous, the number of normal time intervals that are incorrectly predicted to be anomalous, and the number of anomalous time intervals that are incorrectly predicted to be normal, respectively. F1 score is calculated by $2 \times \frac{P \times R}{P+R}$.

We employ the following two widely-used metrics by previous work [18], to evaluate the effectiveness of *Fault Locator*. **Recall of Top-k** ($\boldsymbol{R@k}$) refers to the probability that root causes can be included in the top k results. Higher $R@k$ denotes more effective root cause localization. We choose $R@k$ ($k = 1, 2, 5$) in the experiment. **EXAM Score** ($\boldsymbol{ES}$) refers to the average count of incorrect candidates that have to be excluded manually by operators before localizing the correct root cause. If ES is larger than 10, we set ES as 10.

5 Experimental Evaluation

5.1 Effectiveness Comparison

We use some state-of-the-art trace-based unsupervised approaches to validate the performance of *MicroSketch* on anomaly detection and RCA, including MicroRank [18], tprof [6] and TraceAnomaly [10]. Note that we assume that all the anomalies have been detected before RCA.

Anomaly Detection. Table 3 compares the overall performance of anomaly detection and lists the obtained result with the best F1 score. *MicroSketch*,

Table 3. Comparisons of *MicroSketch*'s *anomaly detector* and baselines.

Dataset	Approach	F1 score	F1 score impr.	Precision	Precision impr.	Recall	Recall impr.
$\mathcal{A}$	*MicroSketch*	**0.925**	–	**0.93**	–	**0.92**	-
	MicroRank	0.834	↑ 10.9%	0.84	↑ 10.7%	0.829	↑ 11.0%
	tprof	0.413	↑ 124.0%	0.327	↑ 184.4%	0.493	↑ 86.6%
	TraceAnomaly	0.804	↑ 15.0%	0.823	↑ 13.0%	0.786	↑ 17.0%
$\mathcal{B}$	*MicroSketch*	**0.934**	–	**0.877**	–	**1.0**	–
	MicroRank	0.865	↑ 8.0%	0.90	↓ −2.6%	0.833	↑ 20.0%
	tprof	0.545	↑ 71.4%	0.48	↑ 82.7%	0.631	↑ 58.5%
	TraceAnomaly	0.804	↑ 16.2%	0.70	↑ 25.3%	0.946	↑ 5.7%

Table 4. Comparisons of *MicroSketch*'s *fault locator* and baselines

Dataset	Approach	R@1	R@1 Impr.	R@2	R@2 Impr.	R@5	R@5 Impr.	Exam Score
$\mathcal{A}$	*MicroSketch*	**0.96**	–	**0.96**	–	**1.0**	–	**0.16**
	MicroRank	0.98	↓ −2.0%	0.98	↓ −2.0%	0.98	↑ 2.0%	0.2
	tprof	0.64	↑50.0%	0 .64	↑ 50.0%	0.70	↑ 42.9%	3.12
	TraceAnomaly	0.62	↑ 54.8%	0.70	↑ 37.1%	0.86	↑ 16.3%	1.98
$\mathcal{B}$	*MicroSketch*	**1.0**	–	**1.0**	–	**1.0**	–	**0.0**
	MicroRank	1.0	0.0%	1.0	0.0%	1.0	0.0%	0.0
	tprof	0.75	↑ 33.3%	0.75	↑ 33.3%	1.0	0.0%	1.0
	TraceAnomaly	0.875	↑ 14.3%	0.875	↑ 14.3%	0.875	↑ 14.3%	1.125

MicroRank and TraceAnomaly achieve over 0.8 in F1 score. However, *MicroSketch* achieves the best result on both $\mathcal{A}$ and $\mathcal{B}$ with an average of 40.9% improvement in F1 score. The F1 score of *MicroSketch* outperforms the compared unsupervised approaches by 10.9%∼124% on $\mathcal{A}$ and by 8.0%∼71.4% on $\mathcal{B}$. tprof performs poorly because tprof detects anomalies using simple ratio relationships.

Root Cause Localization. Table 4 compares the overall effectiveness of RCA. The R@1 results of *MicroSketch* on $\mathcal{A}$ and $\mathcal{B}$ are 0.96 and 1, respectively. *MicroSketch* achieves an average of 25.0% improvement in R@1. The ES of *MicroSketch* achieves 0.16. MicroRank works better in RCA since MicroRank fully leverages PageRank and Spectrum technology and takes a lot of time to get a convergent result. tprof intuitively believes that the more times an operation is called and the longer time it takes, the more anomalous it is. In the operation and maintenance phase, the uncommon pattern should be more concerned rather than the time-consuming pattern. TraceAnomaly analyzes root causes by one specific anomalous trace rather than combining all available traces.

5.2 Adaption

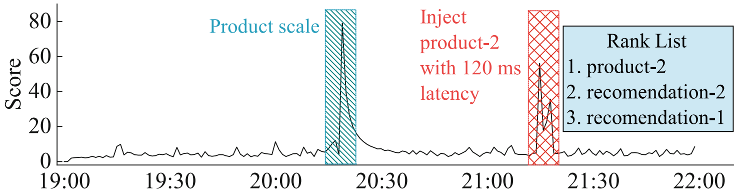


Fig. 7. The anomaly score varies from 19:00 to 22:00 about three hours in HipsterShop. At 20:17, product service instances increase from 2 to 3. This is shown in blue slash shadow. At 21:14, we inject product-2 instance with 120 ms latency and this is shown in the red grid shadow. (Color figure online)

Figure 7 demonstrates the adaptability of *MicroSketch* to changes in system topology. In Fig. 7, the topology of Hipster-Shop changes due to the product service's auto-scaling at 20:17. *MicroSketch* perceives that the pattern of trace data is out of the way and gives the system a high anomaly score at 20:18. Since the topology change is stable, *MicroSketch* adapts to the new pattern and the anomaly score gradually returns to normal again. At 21:14, we actively inject a latency fault to product-2, one of the instances of product service. At 21:15, *MicroSketch* successfully detects anomaly and localizes the root cause (product-2). *MicroSketch* also owns the ability to adapt to other forms of service changes, such as service update.

Table 5. The overhead of the entire *MicroSketch* and single modules of *MicroSketch*.

Module	CPU utilization (%)	Memory usage (MB)	Time(s)	Note
MicroSketch	12 ± 2	200 ± 20	1.1 ± 0.3	10000 traces
Status encoder	12 ± 2	170 ± 10	0.9 ± 0.2	10000 traces
Anomaly detector	12 ± 2	180 ± 10	0.2 ± 0.1	1 time interval
Fault locator	12 ± 2	120 ± 10	0.001	1 anomaly

5.3 Overhead

Table 5 shows the overhead of various modules of *MicroSketch*. *Status Encoder* consumes about 12% CPU utilization, 170 MB memory and 0.9 s to encode 10000 traces as status vector. *Anomaly Detector* takes about 12% CPU utilization, 180 MB memory and 0.2 s to detect whether a vector is anomalous or not. *Fault Locator* spends very little time which is smaller than 0.001 s and consumes 12% CPU utilization and 120 MB memory. The whole *MicroSketch* costs about 12% CPU utilization, 200 MB memory and 1.1 s to analyze 10000 traces. Compared to the overhead of other baselines in Table 1, *MicroSketch* reduces the memory usage by about 50% and is at least 60x faster. *MicroSketch* is more lightweight because *MicroSketch* exploits two efficient data structure DDSketch and RRCForest with a low complexity.

Status Encoder' space complexity is sublinearly related to the number of traces in the time interval, and the time complexity is linearly related to the number of traces in the time interval. *Anomaly Detector*'s space complexity is linearly related to the product of *tree size* and *tree number*, and the time complexity is sublinearly related to the product of *tree size* and *tree number*.

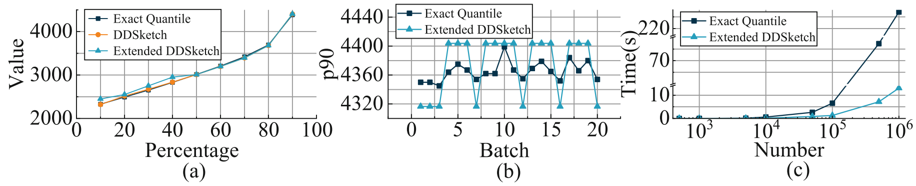


Fig. 8. Comparisons of exact quantile, DDSketch and extended DDSketch. (a) The exact quantiles vs. the values estimated by DDSketch and extended DDSketch. (b) The exact p90 vs. the estimated p90 of a data stream (20 batches of 100,000 values). (c) The consuming time of exact quantile and extended DDSketch.

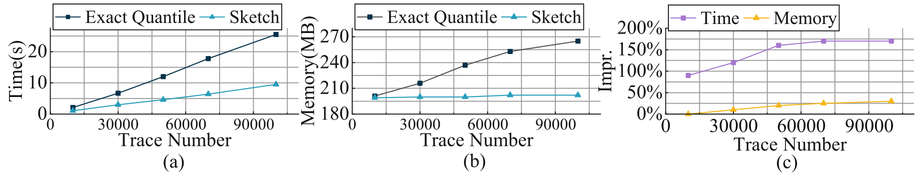


Fig. 9. Comparisons of exact quantile and sketch technology. (a) The time using exact quantile and the time using sketch. (b) The memory usage using exact quantile and the memory usage using sketch. (c) The improvements that sketch brings.

5.4 Sketch Technology

Efficiency and Error. Figure 8-a shows that extended DDSketch has the same relative-error guarantees as DDSketch on the tail data. However, extended DDSketch reduces bucket usage at the cost of losing the relative-error guarantees on the head data which we barely focus on. We employ extended DDSketch on 20 batches of 100000 values to calculate the p90 and the result is shown in Fig. 8-b. The estimated p90 always keeps relative-error guarantees. The relative-error guarantees ensure that the estimated quantiles can be used for the following modules. We implement quicksort to calculate exact quantiles. Figure 8-c shows the consuming time of exact quantile and extended DDSketch. The time of calculating the estimated value is much less than the exact value when the number of data increases.

Ablation. For *MicroSketch*, sketch technology is not indispensable. We remove the sketch technology from *Status Encoder* and use exact quantile instead of it. We analyze various numbers of traces in the time interval. Figure 9-a and 9-b show that the overhead of exact quantile rises dramatically as the number of traces increases, but the rise of sketch technology is relatively flat. Figure 9-c presents that the sketch technology achieves 170% improvement on time and 25.0% improvement on memory usage by analyzing 100000 traces. Thus, *MicroSketch* can scale up readily in large microservice systems.

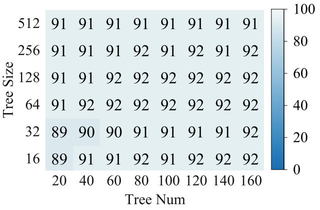


Fig. 10. F1 score using various parameters.

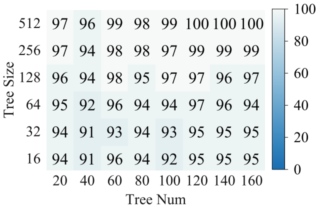


Fig. 11. R@1 using various parameters.

5.5 Sensitivity

Tree Size and Tree Number. *Tree size*, which determines how many vectors RRCTree maintains, is a key parameter for our model. *Tree number* means how many RRCTrees *MicroSketch* creates and also is significant. We set the different values for these two parameters and conduct experiments on $\mathcal{A}$. Figure 10 shows that the difference between the maximum and minimum values of F1 score is 3%. Figure 11 shows that larger parameters can achieve a better result on RCA. However, non-optimal parameters also work well and achieve 91%-95% in R@1. In conclusion, *MicroSketch* is not sensitive to these two parameters.

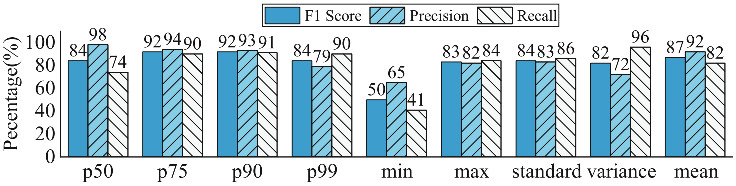


Fig. 12. The performance of our model using various statistics.

Statistical Magnitude. We replace the p90 in the status vector with other statistics. Figure 12 presents that different statistics have different effects. The maximum and minimum values do not work well because of the system jitter. Other simple statistics, such as mean, standard deviation and variance, are easily influenced by a few extremums and lack the ability to perceive issues that slightly affect only part of the requests. Therefore, quantile is a splendid statistic for profiling data. Specific quantile forms specific feature. In practice, it is essential to apply various key quantiles simultaneously in *MicroSketch*.

6 Discussion

MicroSketch forms sketch-based features for anomaly detection and combines the information provided by all anomalous invocations for root cause localization.

Therefore, *MicroSketch* keeps its effectiveness. However, there are some limitations. Firstly, *MicroSketch* focus on the detection and localization of performance issue, so it is helpless over the faults which manifest in other forms. Secondly, *MicroSketch* relies on trace data. The credible trace architecture of microservice systems is an important part to ensure the effectiveness of the method.

7 Related Work

Anomaly Detection. Both TraceAnomaly [10] and Nedelkoski [12], use deep learning method to learn normal patterns of traces offline and detect anomalous traces online. They are useful to detect trace anomalies. However, they require a long time to train the model. Further, when the microservice system changes, they have to retrain the model. Compared to them, *MicroSketch* does not need training and owns the ability to adapt to the system without any human intervention. Seer [3] leverages deep learning to learn spatial and temporal patterns with the KPIs of each service. Hora [13], based on monitored time series metrics, combines architectural knowledge with Bayesian networks to determine the occurrence of performance issues. Microscope [8] detects anomalies by comparing the KPIs with the SLOs of the application. Fully leveraging various types of KPIs, these methods can detect more comprehensive anomaly types. Instead, *MicroSketch* focuses on detecting performance anomalies and localizing root causes more efficiently and effectively.

Root Cause Localization. Zhou [21] designs a trace visualization tool, which allows application operators manually analyze anomalous traces. This tool is very practical but labor-intensive because of the large scale of traces. While *MicroSketch* provides automatic anomaly diagnosis and RCA. MicroRank [18] analyzes clues provided by normal and abnormal traces and utilizes spectrum techniques to localize root causes. tprof [6] hierarchically groups traces by request type and trace structure and calculates increasingly detailed aggregated statistics. These two methods spend a lot of time on obtaining fine-grained and convergent localization results. Compared to them, *MicroSketch* is at least 60x faster and more suitable for large-scale systems. As the number of traces grows, *MicroSketch* will be more advantageous. Many RCA methods are based on KPI, such as MonitorRank [7], Sieve [17] and CauseInfer [1]. MonitorRank [7] forms a system topology graph and uses the personalized PageRank algorithm to determine possible root causes. Sieve [17] reconstructs the system topology and infers possible root causes by representative KPIs. CauseInfer [1] builds a two-layered hierarchical causality graph and uses statistical methods to infer root causes. *MicroSketch* utilizes traces, which carry request information about invocation paths and latency of these invocations, to acquire an API-level system topology that helps to precisely localize root causes.

8 Conclusion

This paper presents *MicroSketch*, an unsupervised lightweight approach to detect performance issues and localize root causes in microservice environments via Sketch-based features and adaptive RRCForest. *MicroSketch* can adapt to changes in microservice systems. The experimental evaluation demonstrates the efficiency and effectiveness of *MicroSketch*. Moreover, *MicroSketch* is at least 60x faster than other methods in terms of diagnosis time. In practice, *MicroSketch* overcomes the challenges imposed by the large scale of traces and the dynamic of microservices, and can scale up readily in large microservice systems.

Acknowledgements. The research is supported by the National Key Research and Development Program of China (2019YFB1804002), the National Natural Science Foundation of China (No. 62272495, 61902440), the Basic and Applied Basic Research of Guangzhou (No. 202002030328), and the Natural Science Foundation of Guangdong Province (No. 2019A1515012229). The corresponding author is Pengfei Chen.

References

1. Chen, P., Qi, Y., et al.: Causeinfer: automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems. In: INFOCOM 2014, pp. 1887–1895. IEEE (2014)
2. Dragoni, N., et al.: Microservices: yesterday, today, and tomorrow. In: Present and Ulterior Software Engineering, pp. 195–216. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67425-4_12
3. Gan, Y., Zhang, Y., et al.: Seer: leveraging big data to navigate the complexity of performance debugging in cloud microservices. In: ASPLOS, pp. 19–33 (2019)
4. Gao, K., Sun, C., et al., S.W.: Buffer-based end-to-end request event monitoring in the cloud. In: NSDI 22, pp. 829–843. USENIX Association (2022)
5. Guha, S., Mishra, N., et al.: Robust random cut forest based anomaly detection on streams. In: ICML, pp. 2712–2721. PMLR (2016)
6. Huang, L., Zhu, T.: tprof: performance profiling via structural aggregation and automated analysis of distributed systems traces. In: SoCC 2021, pp. 76–91. ACM (2021)
7. Kim, M., Sumbaly, R., et al.: Root cause detection in a service-oriented architecture. ACM SIGMETRICS Perform. Eval. Rev. **41**(1), 93–104 (2013)
8. Lin, J., Chen, P., Zheng, Z.: Microscope: pinpoint performance issues with causal graphs in micro-service environments. In: Pahl, C., Vukovic, M., Yin, J., Yu, Q. (eds.) ICSOC 2018. LNCS, vol. 11236, pp. 3–20. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03596-9_1
9. Liu, F.T., Ting, K.M., et al.: Isolation-based anomaly detection. TKDD **6**(1), 1–39 (2012)
10. Liu, P., Xu, H., et al.: Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks. In: ISSRE 2020, pp. 48–58. IEEE (2020)
11. Masson, C., Rim, J.E., et al.: DDSketch: a fast and fully-mergeable quantile sketch with relative-error guarantees. Proc. VLDB Endow. **12**(12), 2195–2205 (2019)
12. Nedelkoski, S., Cardoso, J., Kao, O.: Anomaly detection from system tracing data using multimodal deep learning. In: CLOUD 2019, pp. 179–186. IEEE (2019)
13. Pitakrat, T., Okanović, D., et al.: Hora: architecture-aware online failure prediction. JSE **137**, 669–685 (2018)
14. Shkuro, Y.: Mastering Distributed Tracing: Analyzing performance in Microservices and Complex Systems. Packt Publishing Ltd, Birmingham (2019)

15. Sigelman, B.H., Barroso, L.A., et al.: Dapper, a large-scale distributed systems tracing infrastructure. Google, Inc, Technical Report (2010)
16. Soldani, J., Tamburriand, et al.: The pains and gains of microservices: a systematic grey literature review. J. Syst. Softw. **146**, 215–232 (2018)
17. Thalheim, J., Bhatotia, P., et al.: Cntr: Lightweight {OS} containers. In: 2018 USENIX, pp. 199–212 (2018)
18. Yu, G., Chen, P., et al.: Microrank: end-to-end latency issue localization with extended spectrum analysis in microservice environments. In: WWW 2021, pp. 3087–3098. ACM / IW3C2 (2021)
19. Yu, G., Chen, P., Zheng, Z.: Microscaler: automatic scaling for microservices with an online learning approach. In: ICWS 2019, pp. 68–75. IEEE (2019)
20. Yu, G., Chen, P., Zheng, Z.: Microscaler: cost-effective scaling for microservice applications in the cloud with an online learning approach. IEEE TCC **10**(2), 1100–1116 (2022)
21. Zhou, X., Peng, X., et al.: Fault analysis and debugging of microservice systems: industrial survey, benchmark system, and empirical study. TSE **47**(2), 243–260 (2018)