Analysis of Root Cause Analysis Tools for Cloud Applications

Nikhil Sheoran, Nikunj Gupta
University of Illinois at Urbana-Champaign
{sheoran2,nikunj}@illinois.edu

Abstract

With the ever increasing complexities of modern day applications, monitoring metrics and root-cause analysis (RCA) tools are of prime importance. Automated RCA tools are hard to implement given the unpredictable system behavior and complicated dependency structure of microservices. Furthermore, faults arising due to failures on multiple microservices are hard to analyze and localize. In this report, we explore two automated RCA tools, namely Sieve and MicroRCA. Performance is measured using the SockShop microservices benchmark and the results are reported. Finally, we present an outlook on various pitfalls and corresponding improvements on the RCA tools explored.

1 Introduction

Troubleshooting distributed systems is extremely difficult. The system behavior is unpredictable, dependencies are complex and source of problems are varied. The current monitoring tools - including logging, metrics and tracing - aim to capture the state of the system to be able to later diagnose and debug a certain fault helping in performing root cause analysis (RCA) of the problem. Monitoring and RCA are thus two important tools to ensure system reliability. The goal of a monitoring tool is to look for symptoms of a fault. Once a fault is detected, RCA is used to find the cause(s) of the fault. However, the process of root-cause analysis is largely manual, leading to loss in valuable system uptime.

The complex nature of interdependencies across different microservices as well as across infrastructure and application level metrics makes the root cause identification process time-consuming. In addition, a fault due to multiple causes is difficult to detect because each of the causes might not be capable of generating the fault on its own. This necessitates the existence of an automated RCA tool. Various recent works [14–18] have tried to solve this problem. In this report, we will look at the performance of these tools on a variety of fault scenarios. We will compare their performance and identify the gaps in the existing RCA tools. Collecting trace data requires significant application instrumentation and capturing contexts. Monitoring metrics, specially RED metrics, requires very minimal instrumentation. Service mesh tools like *istio* [3] can automatically provide observability on inter-service traffic. Hence, we specifically look at tools that utilize monitoring data to perform root cause analysis.

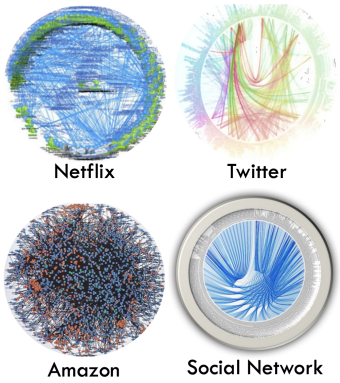

Figure 1. Complexity of Microservices

2 Background

We will first look at a few basic concepts that would be used throughout the report.

Microservices: A microservice is a small application that performs a specific job. These are independently maintained, tested and deployed. Each microservice is deployed independently in different containers (at the same or different cluster nodes). For the context of this report, we deployed individual microservices as different containers in the same machine. A large service is usually composed of 100s to 1000s of microservices. Figure 1 (taken from Gan et al. [12]) shows an example of the complex dependency between the various microservices deployed at major companies.

Logging: The developer of an application program can add events along with system state information to an application. These events get recorded as timestamped logs during the execution of the application. During debugging a failure, one can refer to these events to go into more detail of the exact flow of the request. Log data is usually unstructured and has high volume, making automated solutions less common. For this reason, we do not consider RCA solutions primarily utilizing log data in this report.

Metrics: Metrics represent the raw measurements of resource usage or performance that are observed and collected throughout the system. These can be at the infrastructure level - for example CPU usage, memory usage, disk usage, network bandwidth usage, etc. or at the application performance level - for example, latency, error rate, throughput,

etc. The RED metrics - representing *Rate* (the number of requests per second), *Errors* (the number of failed requests) and *Duration* (the time taken by each request) are common metrics typically collected for any instrumented service. These metrics are typically collected along various dimensions, for example: service, request url, request method, response code, etc. Various alerts (to monitor significant changes in metrics) and dashboards (to get a quick view of the system health) are defined on top of these metrics. In this report, we focus on RCA tools that utilize the metrics' data to identify the root cause.

Tracing: Tracing enables one to observe a request as it propagates through different components of the system. Each request is tagged with a unique identifier. Each interaction is organized as a span. The spans follow a hierarchical structure where a span (i.e. an event) can have multiple child spans (i.e. dependent events). This hierarchical structure provides a much finer view in the request propagation. The OpenTracing Project [8] developed standard specifications for collecting tracing data that could be used across various vendors and tools. Recently, OpenTracing was archived in favor of OpenTelemetry [7]. OpenTelemetry provides a collection of tools, APIs and SDKs to instrument, generate, collect, and export telemetry data (metrics, logs, and traces). Typically, RED metrics could be directly generated from span tracing data using some tools. Metrics Generator in Tempo, Grafana [2] is one such tool. OpenTelemetry is a recently adopted standard and a lot of microservices with tracing instrumentation were instrumented with OpenTracing. The metric generator tools that we came across had limited support for OpenTracing data and hence couldn't reliably generate metric data.

Service Dependency Graph: A service dependency graph represents how a microservice depend on other microservices of the application. Such a dependency graph coupled with monitoring data allows one to reason about potential dependency related failures. It also helps in identifying potential impacts of any outage. Such a graph is a key input for causality-based RCA methods, as it provides an initial causal structure. A static dependency graph can be built based on the application logic. A dynamic dependency graph can be obtained based on tracing data. The differences between these two dependency graphs can also help identify unwanted dependency edges.

3 Related Work

There has been significant work recently in the domain of automated Root Cause Analysis for cloud services. In this section, we explore various such industrial and academic research.

WatchDog[1] is an algorithmic feature by DataDog. It observes trends in the APM and Infrastructure metrics. It looks for irregularities in these metrics for a long period of time and generates a cohesive story based on these irregularities. These stories focus on understanding the anomalous behavior metric and suggesting possible next steps, but fails to uncover the root cause. Thus, WatchDog is more of an automatic alerting tool to identify unusual behavior in a metric. It does try to group together different related behaviors in that timeframe to give a better understanding of the relevant context, but it doesn't surface automatic insights on possible root causes.

Canopy [13] is an end-to-end tracing and analysis system that has been deployed in production at Facebook for more than 2 years. It allows for a flexible multi-domain tracing system that consumes and analyzes aggregates over trace data. Canopy is based on the low-level trace data (by following a request through the different systems using a requestID), constructs modeled traces capturing the causality between the processes and the machines (since it follows the request path). But the major goal of Canopy is not to surface automatic insights instead to allow human-generated hypotheses to be tested quickly. Thus, the onus of quickly identifying root causes still lies with the engineer.

Sieve [15] performs clustering on the set of available metrics for each component (or service) and chooses a representative metric from each of these clusters. Based on a pairwise Granger Causality test, it determines edges between these representative metrics (across components). Sieve only looks at the newly added or discarded representative metrics in order to identify novel components. Now, these representative metrics are significantly dependent on the quality of the clustering and the chosen distance function. Furthermore, Sieve also fails to capture the causal relationships between a representative metric and the metrics belonging to that cluster.

MicroRCA [17] works under the assumption that metrics having high correlation with the faulty metric has a higher probability of being the root cause. It uses BIRCH clustering to find the anomalous latency metrics and build an anomalous graph (using call graph). It then gives edge weights based on anomaly detection confidence (a heuristic based threshold), correlation between service's response time and container metrics. It then uses a graph centrality algorithm called PageRank to identify the root cause of the fault. Note, MicroRCA requires setting heuristic based threshold as well as applies to cases when the fault manifests into an increased response time. Not all faults might result into an increased response time (ex. increased error rate due to a dependent service's failure) and MicroRCA fails to capture such faults.

[1]Watchdog: https://docs.datadoghq.com/watchdog/

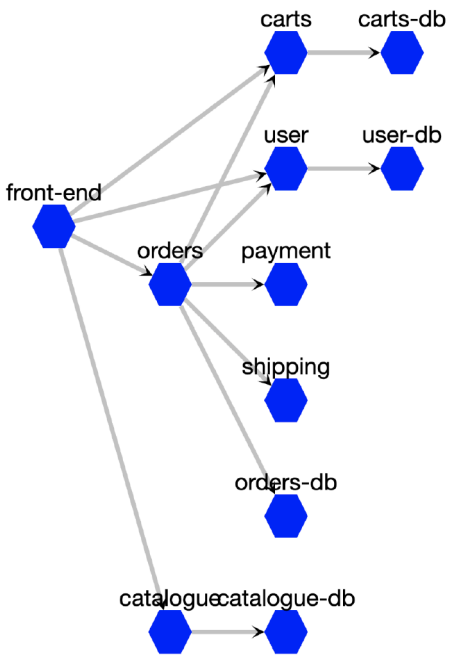

Figure 2. Microservices and Dependencies in *SockShop*.

4 Choice of Microservice Application

DeathStarBench suite [11] includes three end-to-end services made up of 10s of microservices. The microservices are written in different languages and programming models. The applications are instrumented with a lightweight distributed tracing system that tracks requests at RPC granularity. We analyzed and set up the three microservices available in DeathStarBench repository [1]. The three microservices are instrumented with Jaeger using Jaeger-Client [5] that uses OpenTracing specifications. Jaeger-Client now has been deprecated in favor of OpenTelemetry [7]. But unfortunately, we could not find tools that converted OpenTracing Jaeger to monitoring metrics. The tools that we found either required RPC-based OpenTracing (ex: Jaeger Analytics Java [4]) or required OpenTelemetry based Jaeger tracing (ex: Tempo, Grafana [2]).

Hence, we identified another microservice demo application called *SockShop* [10]. *SockShop* represents the user facing part of an online shop with 13 different microservices. Figure 2 shows the different microservices in *SockShop*. *SockShop* can be deployed using common orchestration frameworks like Docker, Kubernetes, Mesos, etc. *SockShop* is instrumented to collect monitoring metrics data. The application also has various Grafana dashboards that can be used to visualize the metrics' data.

5 Experiments

In this section, we will look at the various experiments we performed and the results we obtained.

5.1 Experimental Setup

We deployed SockShop using `docker-compose` on a single machine. The application was deployed with a replication factor of 2 (for all the services) thus tolerating one crash fault (of the container). The experiments were run on an Intel Xeon CPU. The load tests were run using Locust [6], a load testing tool to simulate user workload. The load test framework allows us to vary the total number of requests and the number of simulated clients.

5.2 Root Cause Analysis Tools

MicroRCA: We extract the latency and infrastructure metrics for the different microservices. BIRCH clustering is performed on these obtained metrics. The dependency graph shown in Figure2 is then used to compute the anomalous subgraph. PageRank of root cause metrics is then obtained based on the anomaly score. We evaluate MicroRCA by varying the faulty edge weight parameter in the range of 0.15 - 1.0 in steps of 0.05 and the threshold in the range of 0.01 to 0.1 in steps of 0.005. The value of threshold and faulty edge weight is chosen to maximize the prediction.

Sieve: Sieve requires us to provide both non-faulty and faulty data. First, k-shape based clustering is run on the metrics to obtain representative metrics of the cluster. Once we have the representative metrics, the causal dependency is identified by using Granger Causality test. Based on the two graphs, a similarity and novelty measure is used to identify the metrics that corresponds to the changes between the two versions of the application.

5.3 Data Generation Process

We injected various types of faults in different microservices of SockShop and generated metrics' data corresponding to each of the scenario. More specifically, to generate the data, we follow the following steps:

- The *service is started* with all containers appropriately running with a replication factor.
- The *fault is injected* using a separate fault injection script. The fault injection takes a specific microservice and injects the specified fault in that microservice's docker container. This provides us with a *true label* for the root cause of the fault.
- A *user workload is run* against the faulty system. The start and end time of the user workload are noted.
- The *data for the metrics* generated between this duration are extracted from Prometheus and stored as the input data for the RCA tools.

Once we have generated the input data along with a true root cause label, we run it through the RCA tools. We compare the output of the RCA tools against this true label to analyze the performance of the RCA tool. This process is done for all the different fault types for each microservice in *SockShop* generating a large number of datasets.

5.4 Fault Injection Scenarios

To inject a fault we use Pumba [9], a chaos testing tool for Docker containers. We inject four types of faults in our system.

- **Crash:** The *kill* command of Pumba is used to kill a specific container.
- **CPU Contention:** The *stress* command of Pumba is used to stress test a docker container. It uses `stress-ng` to run stressors in the docker container.
- **Memory Contention:** The *stress* command of Pumba is used for memory contention faults. The memory stressors of `stress-ng` are used for this.
- **Network Delay:** The *netem* command of Pumba is used to inject network faults. The *delay* sub-command enables us to add randomized delays to the egress traffic of the specified docker container.

5.5 Results

We evaluate the two RCA tools on the obtained monitoring metric data. The SockShop benchmark is executed for 30 mins and the metrics obtained are used for our results. Each fault is injected separately in 11 microservices. We compare Sieve against MicroRCA using the **precision at top-k** metric, which denotes the fraction of faults for which the root-cause was correctly identified in the top k results. If the faulty service (and the relevant metric) is present in the top-k, it's considered a 1 otherwise 0. Precision is then the fraction of 1 over the total cases.

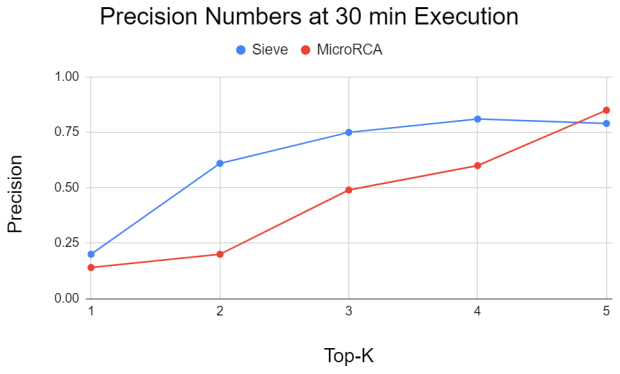

Figure 3. Top-K performance of Sieve and MicroRCA

Figure 3 highlights the performance characteristics of Sieve and MicroRCA. At top-1 condition, we see Sieve slightly outperform MicroRCA. However, both Sieve and MicroRCA are not able to deterministically localize the faults correctly. The precision score gets significantly better for Sieve at top-2 scenario. MicroRCA improves at higher k but still lags behind Sieve. Our hypothesis for the same is as follows. Sieve first reduces the noise by filtering out unvarying metrics followed with differences in non-faulty vs faulty call graphs. This allows Sieve to better reason about the root cause. Furthermore, MicroRCA relies on the correlation of faulty metrics with root cause metrics which might not always be the case. At top-5 condition, we see MicroRCA marginally beating Sieve at correctly predicting. The difference is not significant enough to draw conclusions on their performance.

6 Discussion on Potential Improvements

In this section, we discuss additional fault injection analysis and potential improvements that can be made to the existing RCA tools.

Additional Analysis: In this report we primarily focused on detecting root cause when a single-fault was injected. Various other interesting fault scenarios can be injected and the performance evaluated.

Multiple faults can occur together and make the detection process difficult. In the future, we can inject multiple faults and try to detect multiple faults using these RCA tools. Also, the existing faults were injected agnostic of the service.

A more thorough analysis would be to inject faults specific to the microservice, targeting *specific routes* of the microservice. For example: consider an *orders* microservice that allows one to create a normal order and a one-day delivery order. A fault could be injected in a dependency which is specific to the one-day delivery order route. Owing to this specific dependency, the overall orders microservice's metrics might not show significant deviation unless broken down by the order type. Uncovering such a fault through existing tools that rely on aggregated metrics could be a challenging and interesting problem.

Another major reason for faults is a significant change in the *user workload*. For example, consider a scenario where a deployed change led to the number of requests being sent to a dependent microservice increase by a significant factor. Due to such drastic change in the workload behavior, other performance metrics might show significant anomalies and lead to false positive. We can inject such faults by changing the composition and scale of the user workload.

Request Specific Dependency Graph: When we looked at the metrics collected, we observed that these metrics have multiple different dimensions associated with them. For example: there are various dimensions like request method (ex: GET, POST, DELETE, etc.), request path, status code (ex: 200, 404, 500, etc.). The different routes of the same service might

have different dependency structure. Some microservice dependencies present on one route might not be present on the other route and vice-versa. Thus, taking into account these additional dimensions, can help in further refining the dependency structure. Using a more specific dependency structure can potentially help in performing better RCA.

Sub-Second Root Cause Analysis: RCA tools heavily rely on monitoring metrics from the involved tools (e.g., istio). These tools are not designed for sub-second monitoring intervals. This can result in cases where anomalies propagate faster than the monitoring interval. Under such scenarios, these anomalies will not be detected by any response based RCA (MicroRCA) or interval based evolving RCA (Sieve).

Existing traffic vs Workload generation: RCA tools explored as part of this report requires workload generators to generate monitoring metrics that are then fed to these tools. These RCA tools fail to work on trivial existing traffic and requires a loaded system to correctly function. While a trivial traffic may not expose corner case faults, a trivial traffic would be easier to test. Furthermore, it would also result in a reduction of time as a loaded system metric would no longer be required. To test corner cases, a hybrid approach can help.

7 Conclusion

In this report, we explored the necessity of automated root-cause analysis (RCA) tools with the growing complexity of applications. We analyzed the performance of two chosen RCA tools, namely Sieve and MicroRCA, by running them on the metrics generated from running SockShop benchmark. We observed that Sieve and MicroRCA performs poorly for Top-1 precision numbers when running on metrics generated from 30min execution of the benchmark. At Top-2 to Top-4, Sieve outperforms MicroRCA by a significant margin owing to its better k-clustering algorithm that disregards metrics with low variance. At Top-5, we see MicroRCA marginally beating Sieve in precision scores. The difference is non-significant so no conclusions or evidences were explored on such behavior. Finally, we provide an outlook over the potential improvements on the RCA tools analyzed as part of this report.

8 Metadata

The presentation of the project can be found at:

Zoom Recording Link

The code/data of the project can be found at:

https://github.com/nikhil96sher/cs598-xu-project

References

[1] Deathstarbench. https://github.com/delimitrou/DeathStarBench/.

[2] Grafana tempo. https://grafana.com/docs/tempo/latest/.

[3] Istio:. https://istio.io/latest/.

[4] Jaeger analytics java. https://github.com/jaegertracing/jaeger-analytics-java/.

[5] Jaeger client libraries. https://www.jaegertracing.io/docs/1.33/client-libraries/.

[6] Locust: An open source load testing tool. https://locust.io/.

[7] Opentelemetry. https://opentelemetry.io/.

[8] Opentracing. https://opentracing.io/.

[9] Pumba: Chaos testing tool for docker. https://github.com/alexei-led/pumba.

[10] Sockshop: A microservices demo application. https://microservices-demo.github.io/.

[11] Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., Bruno, A., Hu, J., Ritchken, B., Jackson, B., et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (2019), pp. 3–18.

[12] Gan, Y., Zhang, Y., Hu, K., Cheng, D., He, Y., Pancholi, M., and Delimitrou, C. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems* (2019), pp. 19–33.

[13] Kaldor, J., Mace, J., Bejda, M., Gao, E., Kuropatwa, W., O'Neill, J., Ong, K. W., Schaller, B., Shan, P., Viscomi, B., et al. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th symposium on operating systems principles* (2017), pp. 34–50.

[14] Lin, J., Zhang, Q., Bannazadeh, H., and Leon-Garcia, A. Automated anomaly detection and root cause analysis in virtualized cloud infrastructures. In *NOMS 2016-2016 IEEE/IFIP Network Operations and Management Symposium* (2016), IEEE, pp. 550–556.

[15] Thalheim, J., Rodrigues, A., Akkus, I. E., Bhatotia, P., Chen, R., Viswanath, B., Jiao, L., and Fetzer, C. Sieve: Actionable insights from monitored metrics in microservices. *arXiv preprint arXiv:1709.06686* (2017).

[16] Wang, P., Xu, J., Ma, M., Lin, W., Pan, D., Wang, Y., and Chen, P. Cloudranger: Root cause identification for cloud native systems. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)* (2018), IEEE, pp. 492–502.

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

[18] Zhang, Y., Guan, Z., Qian, H., Xu, L., Liu, H., Wen, Q., Sun, L., Jiang, J., Fan, L., and Ke, M. Cloudrca: A root cause analysis framework for cloud computing platforms. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management* (2021), pp. 4373–4382.