2022 IEEE 15th International Conference on Cloud Computing (CLOUD) | 978-1-6654-8137-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/CLOUD55607.2022.00072

Localizing and Explaining Faults in Microservices Using Distributed Tracing

Jesus Rios
IBM Research
jriosal@us.ibm.com

Saurabh Jha
IBM Research
Saurabh.Jha@ibm.com

Laura Shwartz
IBM Research
lshwart@us.ibm.com

***Abstract*—Finding the exact location of a fault in a large distributed microservices application running in containerized cloud environments can be very difficult and time-consuming. We present a novel approach that uses distributed tracing to automatically detect, localize and aid in explaining application-level faults. We demonstrate the effectiveness of our proposed approach by injecting faults into a well-known microservice-based benchmark application. Our experiments demonstrated that the proposed fault localization algorithm correctly detects and localize the microservice with the injected fault. We also compare our approach with other fault localization methods. In particular, we empirically show that our method outperforms methods in which a graph model of error propagation is used for inferring fault locations using error logs. Our work illustrates the value added by distributed tracing for localizing and explaining faults in microservices.**

***Index Terms*—microservices, distributed tracing, fault localization, failure diagnosis, debugging, root cause analysis**

I. Introduction

Gartner estimates that by 2025 over 95% of new digital workloads will be deployed on cloud-native platforms [1]. To provide high level agility, the cloud platforms become more complex in the face of flexibility with deeper layers of virtualization. The ephemeral nature of containers is advantageous for development, but contributes to challenging SREs task of correctly diagnosing incidents and resolve them timely. Effective response to application failures requires (i) being able to detect that something is wrong with the system from monitoring and probing signals, (i) diagnose the origin of the problem, while maybe deploying some temporary mitigation actions, and, finally, (iii) fixing the system with a permanent solution. This is, in general, labor intensive, time consuming and expensive.

With the advent of cloud and containerization technologies, modern applications are built using cloud-native microservices architectures. Microservices-based architectures break applications into multiple loosely coupled service components that can be developed, tested and deployed independently. This allows for the accommodation of heterogeneous implementation technologies, more scalability, improved resilience, ease of deployment, and achievement of business agility [2]. These properties make microservices ideal for implementing DevOps software principles [3]. However, in practice, identifying the location and cause of an issue in this kind of applications becomes very difficult, especially for architectures with a large number of microservices; thereby, preventing its adoption as exemplified by Segment's decision to go back to Monolith [4]. Traditional monitoring and debugging tools for monolithic applications do not work properly when applied to distributed applications [5]. Quick detection and localization of these failures is therefore key for improving the reliability and availability of production microservice applications. However, current industry debugging practices do not seem to be at a level of maturity to meet this challenge [6].

Logs are useful in debugging the cause of an observed problem in an application. However, searching through log entries can be very time consuming and lead to cognitive overload of the SREs, specially if the logs are distributed across many microservices. In distributed systems, logs are not contextualized and thus it is not possible to understand the flow of events that generated them as requests travel from one microservice to another. Logs are local and only report what happens at a microservice level. We need a global view of the system that can be used to correlate those logs across microservices, for example by specifying the request they belong to. This can be achieved employing modern distributed tracing tools [7]. We use trace data to understand error propagation and to pinpoint the location of the original fault without the need to reproduce the problem.

Fault localization is the process of tracing back failure signals through a distributed application to locate the first failing component. We present here a new approach that monitors in real-time traces and logs from a production application. When errors are detected in a trace a novel fault localization algorithm is automatically triggered to find the faulty microservice that is at the root of the problem. Once found, the logs at such microservice are queried using information from such a trace to narrow down an explanation for the cause of the fault. We demonstrate this approach with an experiment in a well known microservice benchmark and compare our results with other approaches in the literature.

Our contributions include the following.

- We have developed an unsupervised causal inference algorithm to localize faults in microservice-based applications that incorporates domain knowledge on (i) failure propagation, and (ii) the client-server model of communication. The algorithm monitors traces and extracts at runtime a causal model for each problematic trace; thereby, capturing the causal interactions between requests and microservice error relations that is missing in most casual inference approaches to fault localization [8].

- Context-aware log mining. We use the result of the causal inference algorithm to extract the logs that are most relevant to observed request failures. In particular, we extract and present only logs from the service and during the time-window of the span where the fault happened; thereby, significantly reducing the logs required to be debugged by SREs during an ongoing investigation.
- We demonstrate the efficacy of our proposed approach on TrainTicket, a widely used microservice benchmark application, using fault injection experiments. We show that our algorithm always accurately identifies the faulty service; outperforming the approaches reviewed in Section VI.

II. Background

Here we provide background on the fault localization problem and distributed tracing.

A. Fault Localization

Large cloud applications based on microservices architectures are composed of many interdependent microservices. When a problem occurs in one microservice, the error propagates to microservices which directly depend on this service. This has a ripple effect with faults propagating across microservices, creating a storm of failures steaming from all the affected microservices. Finding the exact location of the originating fault is not an easy task due to a vast amount of errors generated by all microservices.

Approaches to fault localization in distributed systems vary. We could classify them based on the observability assumptions each approach makes. Thus, they may consume different monitoring data types, meaning their input may include logs, metrics, traces or a mix of these. Another classification criteria could be their scope. Faults at the application level are not the only type of problem these approaches may considered. Some approaches also focus on localizing application performance issues, infrastructure-layer faults or a mix of these.

In terms of their methodology, most approaches construct a graph modeling the inter-dependencies among the components of the system and use this graph to localise a detected issue. For example, Sage [9] models microservice dependencies with a causal Bayesian networks and uses graphical variational auto-encoders to locate the microservices that have caused a performance issue, detected as a QoS violation. Microscope [10] and MicroHECL [11] use propagation and correlation analysis over a dynamically built call graph to rank root cause candidate locations. Alternatively, MonitorRank [12] uses the call graph to build an unsupervised model for ranking. Some approaches assume knowledge of the dependencies between components in the system, and use this information in their correlation analysis [13]. Others have proposed the use of Granger causality and the Personalized PageRank algorithm over a dependency graph to locate root causes [14]. MicroRCA [15] also uses PageRank for fault localization but after computing anomaly scores for nodes in a graph based on detected anomalies. Although less common, there are also approaches that do not explicitly model causal or dependency relations among system components. For example, supervised learning methods such as those in [16] and [17] have been proposed for directly finding system components causing failures. These methods leverage automatic fault injection frameworks to generate the necessary training data for learning the relation between failure symptoms and root causes.

B. Distributed Tracing

Distributed tracing has grown out of the need for understanding the behavior of applications with distributed architectures. For example, in this kind of applications, every time a user takes an action, requests are propagated through the distributed cloud environment. Tracing provides observability of communication and data as it flows across the many different components of the application.

A trace represents the path of a request through the various services comprising an application. Each trace has a unique identifier and consists of spans. A span represents a logical unit of work, with a start and finish time, carried out by some service component within the application. Each span within a trace has a unique identifier and a possible reference to other related spans. For example, if service A makes a HTTP request or RPC to service B, we have a direct causal relationship between these two services: service A depends on service B to do its work and, indeed, service A must wait for B to respond to successfully complete that work. In other words, a failure in B's operation may cause A's operation to fail. From a tracing perspective, the span representing the work carried out by service B is the *child* of the span representing the work by service A. In this child-parent relation, the span in B is the child span and the span in A is the *parent* span, and the parent span depends on the child span to do its work.

Spans and their *parent* references within a trace form a tree representing the dependencies between microservice in processing requests. The root span of such tree will be the span with an empty parent span reference and typically represents the start of a user transaction in the application.

There are two main methods to generate tracing information in a distributed application: black-box based and annotation-based. Black-box based tracing uses statistical analysis to infer span correlations post-hoc. This method needs vast amounts of data and typically is error prone. Annotation-based tracing methods, such as those popularized in [18], propagate annotated data from one microservice to the next as request travels through the application. The propagated data constitutes the context of a trace and includes information such as globally unique identifiers for the trace and the parent span. This information is preserved for reconstruction of the distributed trace. Trace context propagation requires instrumentation of application code. In the last decade, there have been a lot of progress towards making this instrumentation easier, specially with the appearance of open instrumentation standards as well as multiple vendors solutions for automatic instrumentation supporting the most popular programming languages and frameworks [19]. There are also open-source distributed tracing systems such as Zipkin and Jaeger that can generate, collect and store tracing data for instrumented microservice applications [20].

III. Our Automated Fault Localization Solution

In this section, we outline the design principles and design overview of our Faulty Service Finder (FSF), a fault localization and diagnostics framework to automate the task of detecting application-level faults, identifying their originating location and selecting the logs that may explain the cause of each fault.

A. Design Principles

FSF addresses the challenges of identifying faulty microservices and providing diagnostic information using the following approaches:

1) *Focusing at the application layer to support multi-cloud and hybrid cloud environment.* A key objective of FSF is to ensure that the approach is extensible and usable across different multi-cloud and hybrid cloud environment. Therefore, FSF only gathers telemetry data that is available at the application layer as gathering low-level data from across the full system stack is often not feasible due to organizational and contractual constrains.
2) *Fusing heterogeneous telemetry data for increased observability.* FSF uses telemetry data from across the microservice application, capturing both traces and error logs that are readily available, to increase spatial and temporal observability. The fusion and comprehensive analysis of the data enable both fault localization as well as diagnostics.
3) *Causal inference for explainable fault localization.* FSF uses causality-driven inference technique for explainable fault localization. It uses knowledge of (i) failure propagation, and (ii) the client-server model of communication. FSF extracts the causal model at runtime from each problematic trace; thereby, capturing the causal interactions of requests with microservice's error relations, that are missing in most casual inference approaches to fault localization [8].
4) *Context-aware log mining*. FSF uses the result of the causal inference algorithm to extract the logs that are most relevant to observed request failures. In particular, FSF extracts and presents logs from the localized faulty service and during the time-window of the span where the fault happened; thereby, significantly reducing the logs required to be debugged by SREs during an ongoing investigation.
5) *Unsupervised ML models for dealing with insufficient samples and rare failures.* FSF uses unsupervised ML models and leverages domain knowledge on the system design and architecture to alleviate the challenges of (i) labeling failures, and (ii) acquiring training data on rare failures, especially on rare one-off failures.
6) *Low-cost automation for timely analytics.* The use of unsupervised methods alleviates the need for costly training and re-training of models.

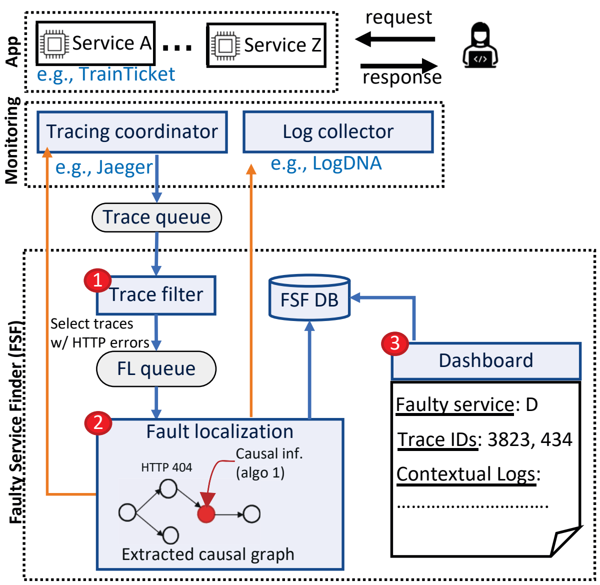

Fig. 1. Fault localization and log selection using distributed traces.

B. Design Overview

Figure 1 shows the overall design of FSF, and its integration with a microservice application. FSF only requires access to the following monitoring capabilities: (i) annotation-based tracing (e.g., Jaeger), and (ii) application logs collection (e.g., LogDNA). Furthermore, we assume that the application has been instrumented to support annotation-based end-to-end distributed tracing at the application-layer. FSF is independent of the underlying container orchestration capabilities, and therefore, is compatible with wide variety of orchestration frameworks such as Kubernetes, OpenShift and Docker Swarm among others.

FSF consists of three services: (i) trace filter (1), (ii) fault localization (2), and (iii) dashboard (3). FSF trace filter acts as interface between the *tracing coordinator* and the *fault localization* service. It actively consumes and processes each trace in the *trace queue* from the *tracing coordinator* in real time, and writes the trace id to the *FL queue* if any of the spans in the trace have HTTP errors. These FIFO (First-In-First-Out) queues are shown in gray boxes in Figure 1 and can be implemented using streaming services or data pipeline services such as Kafka for example. *Fault localization* service consumes data items from the *FL queue*. For each item in *FL queue*, it extracts the trace data by querying the *tracing coordinator* and uses that to estimate the originating location of the fault (i.e., identifies the faulty application service) causing all these errors as well as the parameters necessary to query the logs. Using those parameters, it queries the *log collector* service to extract the relevant logs that could explain why the fault happened in the first place. This approach aims at reducing the number of logs that SREs need to analyze to localize and find the root cause of an issue within a microservice application.

Typically, these kind of applications emit a vast number of logs, sometimes in the order of thousands per second. Reducing the number of logs to focus on, whenever an issue occurs with the application, to a small handful subset, is of great benefit to SREs. Finally, the *fault localization* service writes a tuple consisting of `<trace ID, faulty service, contextual logs>` to the *FSF database*. Services corresponding to the queues and database are not shown in the figure for clarity purposes. Finally, SREs can interact with the *dashboard* service through a web API (or web server) to query and filter the data in *FSF database*. Next, we describe in detail *fault localization* as it is the core service of FSF.

C. Fault Localization

FSF uses Algorithm 1 to identify faulty application services. It applies causal inference principles to the problem of fault localization. The algorithm monitors in real time the traces generated by the system. Using traces as an input, it detects trace-spans containing errors and returns an estimate of the originating fault location as well as a small set of logs to assist with root cause analysis. For each trace, we use the spans' *parent of* references to extract the causal relations between spans in that trace. This constitutes our causal model, which takes the form of a tree and is stored in a data structure that gives us for every span, its parent span in the trace. To build this tree, we just loop through all the spans in the trace and look at their states to find out their parent span. Note that the root span is defined as the only span in the tree with no span parent and therefore with an empty *parent of* value. Spans are identified by their unique IDs given at creation and kept as part of the data carried out in their states.

Given the trace graph, a tree in this case, we can identify the span that started the chain of errors using causal inference. We perform this by looping through all the spans in the trace that have failed, i.e., such that $span \in \textsc{ErrorSpans}(trace)$. From those spans we remove from consideration the ones that have failed children, i.e., such that $\textsc{Children}(span, trace) \cap \textsc{ErrorSpans}(trace) \neq empty$, as their failure can be explained by a children failure. Thus, our algorithm only keeps those spans whose failure cannot be explained by any other span, and therefore they must contain localization information of application failure. This is how we direct causal inference towards finding explainable fault localization information.

If a retained span represents a failed operation carried out by the *server* microservice of an HTTP request, then this span can be considered to be at the root cause of the failure. We consider that a response operation handled by a server span has failed only when a 5xx HTTP status code is associated with the request. For example, a 500 error code would mean that the operation represented by the *server* span could not successfully respond to the client due to an internal failure. If this failure happens in a retained span, no other span exists that may have caused this failure and therefore we consider this to be the fault location. However, if the retained span is a *client* span failing when trying to get a response from a HTTP request with, e.g., a 503 (Service Unavailable) error code, then we conclude that the root cause should be located at the server side of that HTTP call (even when no server span is generated for this request). Thus, if the error at the retained client span comes with a `5xx` status code, we conclude that the fault must be located at the server side of that HTTP request; on the other hand, if the request status code is in the `4xx` class, the error would be caused by the client span itself. A 4xx error code indicates that the request failed due to a fault of the client sending the request. We only mark spans with `4xx` error codes as erroneous if they are *client* spans. A server span with a `4xx` error code is never considered erroneous. Thus, a 4xx error in a retained *client* span cannot be caused by any other erroneous span, and therefore the fault is localized to the retained span's microservice. We mark *client* spans with `5xx` status codes as erroneous for dual purpose: to locate faults when the server side is not available and to make sure error propagation paths follow connected causal chains in the trace tree.

In Algorithm 1, the function $kind(span)$ parses the data in the $span$ to find out whether it operates as a *client* or a *server* of a HTTP request. Similarly, $http.url\_host(span)$ deduces the host's location of the request associated with a $span$, using the host component for the request's URL, and $http.status\_code(span)$ extracts the HTTP response status code. The functions $service\_name(span)$ and $container\_name(span)$ provide respectively the names of the microservice and pod/container associated with a $span$'s service and process.

After localizing a fault, Algorithm 1 retrieve the logs emitted by the service associated with the retained span for the span's duration. To do so, we use the span's start and finish timestamps as well as the identifier of the span process and extract the logs generated by that container between the span's start and finish timestamps. Note that in the case the faulty service is not available, FSF pulls the logs from a parent service. The logs selected by FSF could be used to explain the cause of the fault, and would typically revel the root cause down to the code line. FSF is then able to extract relevant logs automatically when a fault manifests in an application using causal-driven inference to define the log selection criteria.

FSF also needs to alert SREs when failures are detected, so they can take the necessary remediation and repairing actions as quickly as possible. FSF monitors the application in real-time at the request level. Thus, for each fault in the system, every time a request hit the faulty microservice the same fault is detected and localized. Also, the selected logs will be essentially the same. Sending an alert every time this happens will overload SREs with receptive information about the same fault. Alert grouping or silencing methods, such as those available in Prometheus Alertmanager can help with this by simple grouping alerts of similar nature into a single notification or by silencing alerts based on regular expression matches to previously sent active notifications. In the case we have limited resources to address multiple identified faults, we could use the number of traces associated with each fault to prioritize those resources. This give us a proxy for the

Algorithm 1 Fault localization algorithm

```
procedure $\textsc{Main}(trace)$:
  $error\_spans \leftarrow \textsc{ErrorSpans}(trace)$
  for all $span \in error\_spans$ do
    $children \leftarrow \textsc{Children}(span, trace)$
    if $children \cap error\_spans = empty$ then
      $error\_code \leftarrow http.status\_code(span)$
      if $kind(span) =$ “server” then
        $faulty\_ms \leftarrow service\_name(span)$
      else if $kind(span) =$ “client” then
        if $error\_code =$ 4xx then
          $faulty\_ms \leftarrow service\_name(span)$
        else if $error\_code =$ 5xx then
          $faulty\_ms \leftarrow http.url\_host(span)$
        end if
      end if
      print Fault location: $faulty\_ms$
      print $logs \leftarrow \textsc{GetLogs}(span)$
    end if

procedure $\textsc{Children}(span\_target, trace)$:
  $children \leftarrow empty$
  for all $span \in trace$ do
    if $parent(span) = span\_target$ then
      $children \leftarrow children \cup \{span\}$
  return $children$

procedure $\textsc{ErrorSpans}(trace)$:
  $error\_spans \leftarrow empty$
  for all $span \in trace$ do
    $http\_code \leftarrow http.status\_code(span)$
    if ($http\_code =$ 5xx) or
      ($http\_code =$ 4xx and $kind(span) =$ “client”) then
      $error\_spans \leftarrow error\_spans \cup span$
  return $error\_spans$

procedure $\textsc{GetLogs}(span)$:
  $logs \leftarrow \textsc{Query}$(database with logs,
          from: $start\_timestamp(span)$,
          to:    $finish\_timestamp(span)$,
          container: $container\_name(span)$ )
  return logs
```

importance of each fault in term of the number of affected requests. In order to avoid spending effort in low impact faults, we can also establish thresholds that would need to be surpassed before alerting SREs.

IV. Experiment

We demonstrate the effectiveness of our proposed fault localization approach on a publicly available and well known microservice application benchmark. We inject faults into different end points of this application and see whether our fault localization algorithm is able to correctly detect and locate the injected faults. Given the uptake in the use of causal inference to develop algorithms for fault localization [8], we also empirically compare the performance of our algorithm with other approaches based on causal inference. The aforementioned approaches typically reason over an application-level graph, which models causal dependencies between errors occurring at the various microservices, instead of reasoning, as we do, over a request-contingent model of causal dependency associated with each trace. Thus, we have implemented an archetype version of these kinds of approaches and compared its performance with Algorithm 1.

A. Set-up

We set up our experiments on an OpenShift Container Platform (OCP) cluster, which is installed on cloud-based infrastructure. OCP is an augmented Kubernetes distribution developed by RedHat. Our OCP cluster consists of 6 nodes running OpenShift 4.7 version.

Monitoring. We use logDNA to collect application-level logs. We created a logDNA service instance and deployed the logDNA agent in our OCP cluster. The logDNA agent collects logs from all microservices. The logDNA service instance provides an API for searching and exporting the collected logs.

We chose to use Istio service mesh [21] technology along with Jaeger to generate and collect tracing data [20]. We installed in our OCP cluster the RedHat OpenShift service mesh, an Istio distribution adapted to OCP and expanded with some additional features. Jaeger is an open-source distributed tracing tool which is part of this service mesh installation. The Istio proxies that make up the service mesh are configured to send tracing spans data to Jaeger, which in turn will process and centralize this data.

Workload and user-flows. We decided to use Train-Ticket [22], an open-source microservice application benchmark, to test our fault localization approach. We deployed Train-Ticket in our OCP cluster and added the whole application to the service mesh. Train-Ticket consists of 41 microservices simulating a train ticket reservation system in which users can search for trains, book tickets, make payments, collect paid tickets and enter stations to board trains, among other things.

There must be a user request that interacts with the faulty microservice for a fault to be active and manifest. Thus, we created user-flows for Train-Ticket that simulate users' interactions with the application's user-interface, generating a set of cascading requests to the backend. Our user-flows cover 33 of the 41 microservices in Train-Ticket. We have not included in our user-flows the simulation of actions taken by the admin role which is in charge of managing orders, routes, users, stations, trains, prices and so on, leaving as a consequence their corresponding microservices uncovered by our user-flows.

Fault types. We inject the following faults to validate FSF.

(a) *http-service-unavailable* (`HTTP 503`) which mimics a microservice availability failure. In this fault, all requests directed to the injected microservice fail. Such failures occur because of application failures (e.g., hang or crash), infrastructure failures (e.g., pod failure/restart) or deployment issues (e.g., service unable to start).

(b) *http-bad-request* (`HTTP 400`) which mimics request corruption. In this fault, the injected microservice (client) sends a bad HTTP request to a server microservice. Such failures can occur due to malformed request syntax (e.g., a bug), invalid request message framing (e.g., network error) or deceptive request routing (e.g., session hijack).

(c) *http-resource-not-found* (`HTTP 404`) which mimics a resource not found failure. In this fault, the server microservice is available but the resource requested by the client microservice is not found by the server microservice. Such failures occur due to incorrect API specification, bad code, or version upgrade issues.

We use the above three fault types because (i) they cover both clients (`HTTP 400/404`) and server (`HTTP 503`) faults, and (ii) these fault types can occur due to operational, misconfiguration or code issues.

Fault injection. We use Istio's fault injection capabilities [23] to simulate *http-service-unavailable* as Istio can just intercept any request directed to that particular service and return an HTTP 503 error code; thereby, avoiding the need to actually terminate the container in which the microservice is running. We inject *http-service-unavailable* in each of the different microservices covered by our user-flows, except for the microservice associated with the user-interface, for a total of 32 faults.

We inject *http-bad-request* and *http-resource-not-found* by making changes to the application source code and building faulty versions of each modified microservice. We do not use Istio to simulate these two types of faults since intercepting requests before reaching the destination microservice and returning HTTP error codes in the `4xx` class would lead to missing spans and logs. *http-bad-request* and *http-resource-not-found* cannot be emulated for microservices that are at the end of the call chain (i.e., they are leaf nodes in the call chain). Thus, we were able to inject *http-bad-request* and *http-resource-not-found* in only subset of the microservices (18 in total), resulting in 36 faults across both these faults types.

Faults were injected sequentially, one at a time, so there were never more than one injected fault into the application at the same time. During each fault injection experiment, with only one fault injected in one of the microservices, a fixed set of user-flows always executed in the background. Each fault was simulated for a couple of minutes to ensure that the injected fault manifested and impacted the user requests. After each injection, the injected fault was removed from the application before starting another fault injection experiment. At the end of each fault injection period, we collected all the logs and traces generated by the application.

B. Results

Algorithm 1 perfectly detected and located all injected faults, which included *http-service-unavailable*, *http-bad-request*, and *http-resource-not-found*. We note that during the experiments, FSF never produced a false positive, i.e., an estimate of a fault that did not exist in the system, whether injected or internal.

During the time each fault was injected, the subset of logs extracted by our algorithm, with every trace that exhibited span errors, always indicated the presence of the injected fault. The logs also provided information about the failed requests that generated the errors. The specifics of the information depended on the programming language and framework of the microservice sending the request. In the case of `HTTP 400/404 faults`, the logs included more details to help solve the problem. For example, when we injected a *http-resource-not-found* fault in *ts-ui-dashboard*, a Nginx microservice, the selected logs included the (modified) API endpoint of the resource being requested. The fault was injected through source code modification. When the same type of fault was injected, for example, in *ts-ticketinfo-service*, a Java-based web microservice, the selected logs not only indicated a "HttpClientErrorException: 404 Not Found", but also the file and line in the source code where the failed request to *ts-basic-service* took place. There, we could see the URI path modified by the fault injection that generated the error. In section V, we discuss how the explanation obtained from logs can be used to fix the observed request failures.

C. The Value of Distributed Tracing in Causal Inference

To illustrate the value of distributed tracing in our algorithm, we remove traces from consideration. We apply the same causal inference principles as in Algorithm 1 to error logs, assuming we have a causal model of error propagation. Learning this causal model directly from error logs has proven to be very noisy. We instead use the dependency graph as a proxy model of the causal dependencies between errors occurring at the various microservices. This model is easy to estimate in practice by deploying black-box monitoring in each microservice. In our experimental setting, the dependency graph is provided by Istio after running our userflows. We then apply causal inference over this graph to estimate fault locations from the observed logs over each fault injection period. Thus, we assume that the application's causal graph is known.

Thus, we have implemented a fault localization algorithm based only on logs that applies the exact same causal inference principles as in Algorithm 1 but over the application's dependency graph. Fault locations are estimated by eliminating microservices in this graph with error logs that are inferred as being caused by others, until obtaining a set of candidate fault locations. We also implemented an improved version that applies text analytics techniques to reduce fault location estimation sets when they are imprecise, i.e., containing too many possible locations. Fig. 2 shows how the performance of

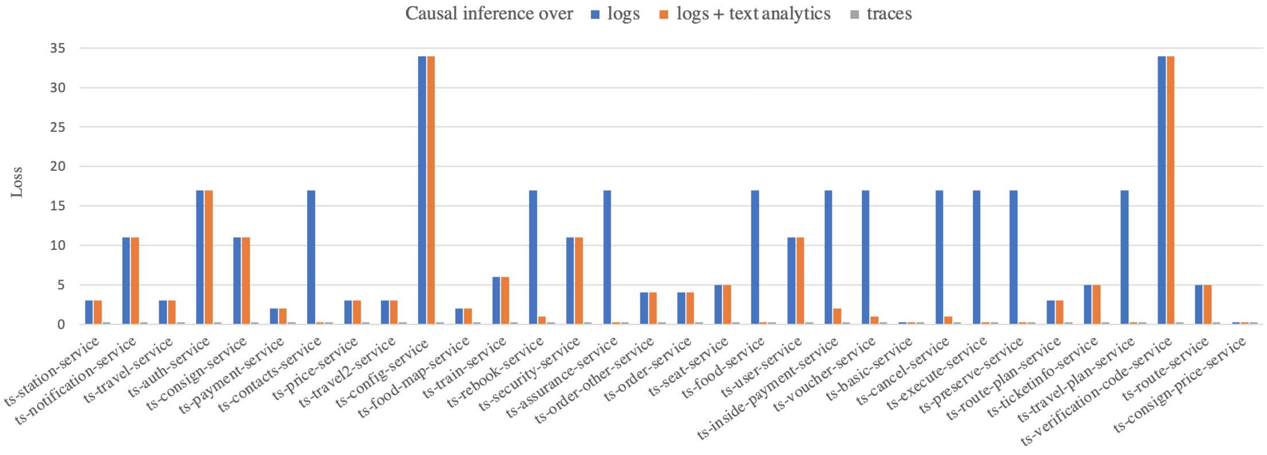

Fig. 2. Comparison (lower loss values are better) of fault localization results for causal inference methods with and without distributed tracing.

Algorithm 1 compares with these other two fault localization algorithms, which do not use distributed tracing.

To measure the algorithms performance, we define the *loss* of a fault localization estimate as the number of incorrect locations in the estimation set. If this set does not contain the location of an actual fault, the loss would be equal to the size of the application topology, i.e., our causal model: in Train Ticket, a total of 34 microservice locations. The topology in this case only includes microservice locations covered by our userflow, since we assume that the investigation is only limited to that part of the application. This loss can be interpreted as the (worst case) number of incorrect locations that an SRE would have to go through to confirm the existence and location of a fault. For example, if the estimate consists of the true location of the only existent fault in the system, the loss of the estimate would be 0: there are no incorrect locations in the set. If the estimation set consists of two locations and only one happens to correspond with the fault in the system, the loss would be 1: the estimation not only contains the location of the true fault, but also another incorrect location that an SRE would have to validate. If the estimation set has n elements and none of these are correct, the loss would be 34, since an SRE would have to go through the n microservices in the estimation set first, and after realizing that none of them are correct, then go (in the worst case) through the complete list of possible locations, 34 in our experiments, to find out where the true fault location is (if there is one in the system).

We computed the loss of each algorithm's location estimate for the same 32 *http-service-unavailable* fault injections in Section IV-B, see Fig. 2. Note that lower loss values mean better estimates. We see that Algorithm 1 clearly outperformed the other two algorithms, which restricted causal inference to reasoning only with logs data instead of traces. The results from the underperforming algorithms are comparable with state-of-the-art approaches that use only log data to locate faults. After evaluating various log-based methods over the same benchmark application, Train-Ticket, and injecting *http-service-unavailable* faults in 17 of its 41 microservices, the best performing method [14] achieved an average loss no better than 8.24. Our straightforward causal inference method over logs resulted in an average loss of 10.84, and of 5.69 when improved with text analytics. Algorithm 1 achieved a perfect average loss of 0.

In Section VI, empirical results from the literature show that current causal inference based approaches fail to perfectly and accurately locate these kind of faults. The failure is due to the use of application-level causal models, such as the dependency graph, obtained through aggregation over many requests. However, causal dependencies between errors in microservices interact with traces. For example, if microservices A and B depend on C, then an error in C propagates to A or B, depending whether the request to C in the underlying trace comes from A or B. If we do not take into account these causal interactions in our inferences, we will not be able to determine causality with certitude, and thus leading to inaccurate fault location estimates. Trace trees are the solution because they provide a different causal model for each trace, and therefore reflect the causal interactions between error propagation and requests in a trace.

V. Experience with diagnosis and mitigation.

In the following section, we describe our experience in using FSF for localizing and diagnosing the TrainTicket application.

A. FSF Diagnosis Power in Practice

FSF is able to detect, localize and diagnose faults for any application in real-time, as and when they occur, as soon as the application is deployed. This is because FSF uses unsupervised causal inference (see Section III); thereby, requiring no training whatsoever either in staging or production environment. FSF demonstrated this ability when it helped us diagnose and fix internal (non-injected) faults in Train-Ticket the moment we start using it.

In our experiments, we noticed that Train-Ticket application exhibited request errors when we started running the user-flow following a period of inactivity. This erroneous behavior disappeared after a minute or so. For this reason, we always waited for a period of time before starting injecting faults in our experiments. However, FSF picked up the traces during this warn up period and we were able to automatically uncover the root cause of these errors. In particular, FSF located the same type of fault in every microservice connected to a MongoDB database instance. The selected logs in each case explained the faults as HTTP 500 Internal Server Errors caused by a MongoDB Socket Exception raised when the microservice tried to connect to its MongoDB database after a period of inactivity. We searched for a solution to this issue using the details in these logs and found that changing the socket configuration would enable the connection to stay alive, and thus solve the problem.

B. *FSF Explanatory Power*

FSF can enhance the explanative power of logs despite them being noisy due to the lack of standardization (e.g., free-flow use of error explanation as well as language dependent logs) and enforcement (e.g., incorrect labeled severity levels). Moreover, using only logs to localize and diagnose a fault may not be sufficient or efficient.

When logs lack some of the information necessary to identify unavailable microservices, FSF causal inference can find this information in the traces. For example, when a *http-service-unavailable* fault was injected in the *ts-auth-service* microservice, every time the userflow generated a trace with errors, FSF always selected one log coming from the *ts-ui-dashboard* microservice to explain the correctly localized fault in *ts-auth-service*, which did not emit any logs due to its unavailability. This log correctly explained the errors seen in the trace as the consequence of a failed HTTP POST request from *ts-ui-dashboard* to a resource with a URI (Uniform Resource Identifier) specified by its server endpoint /api/v1/users/login, with a 503 status code indicating that the server microservice of that request was unavailable. However, the log by itself did not provide information of the host at the server-side of the request, necessary to identify the unavailable microservice causing the `HTTP` 503 error. The server host information could only be found in the trace data. The client span created to represent the HTTP request from *ts-ui-dashboard* to *ts-auth-service* included the URL of the request, in this case http://*ts-auth-service*:12340/api/v1/users/login. The function *http.url_host* was used by FSF to extract the host component, *ts-auth-service*, identifying the unavailable microservice.

We also noticed during our experiments that many logs selected by FSF have an empty or null value in their *level* attribute, despite being essential to explain the detected faults. In these cases, they were in fact the only logs in the system revealing that `HTTP 503` errors have occurred during the fault injection. We would have expected these logs to come with at least an *error* log level value so they can be easily caught by standard log level rules in monitoring systems. An advantage of FSF in this regard is that it does not depend of the log level attribute to assess the importance of logs when selecting them.

C. *FSF Power to Detect Multiple Faults*

FSF is able to detect and localize multiple independent and concurrently occurring faults. For example, a demonstration of this ability occurred in one our experiments. When we injected a *http-service-unavailable* fault into *ts-verification-code-service*, we noticed that FSF identified two independent concurrent faults: the injected fault (HTTP 503) and another internal fault (HTTP 500) that was estimated to be located in *ts-assurance-service*. Since we did not inject a second fault during that fault injection experiment, this estimate seemed wrong at first. However, further investigation revealed that there was indeed an actual internal fault that already existed in the application in that location.

We were able to reproduce the non-injected accidentally occurring fault without the other one having to be injected into the system by simulating a user searching for trains without first login into the application with his name and password. Our typical user-flow did not reveal this fault before because it always started successfully login the user into the application. Only the presence of the *http-service-unavailable* fault in *ts-verification-code-service* made the user-flow unable to successfully login, triggering the manifestation of the internal fault in *ts-assurance-service*, so it could be detected by FSF during such injection period. In this case, the selected logs identified a HTTP 500 Internal Server Error as the cause of the fault, meaning that *ts-assurance-service* encountered an unexpected condition that prevented it from fulfilling a request from *ts-ui-dashboard*. The information about where the request originated came from trace data and did not appear in the logs. The selected logs also tell the type of exception that caused the fault, indicating the user could not be authenticated due to a malformed and therefore invalid JSON Web Token (JWT) that was rejected. The JWT string was empty and the exception not properly handled. The logs also pointed to the lines in the source code of *ts-assurance-service* where the exceptions occurred. After inspecting these lines of code, we easily realized that a possible solution to fix this fault is to handle this exception by sending the user to the login page every time the exception occurs. This would require a small modification to the microservice source code. This also demonstrates how SREs can leverage FSF to fix faults.

VI. Comparison with other approaches

We review some state-of-the-art systems from the academic literature that are aimed at detecting and locating faults in multi-component applications.

Path-based approaches. Path-based approaches leverage metrics data along with the application topology (known apriori) to detect and localize faults. These approaches require handcrafted heuristics to interpret and design anomaly detection algorithms in order to use metrics data successfully.

Examples of this approach include: MicroHECL [11], MonitorRank [12], and Microscope [10]. MicroHECL is a root cause localization system, in which observability is limited to metrics such as inbound and outbound service calls and response times for each application component. Every time MicroHECL detects an anomaly such as a performance, reliability or traffic issue, it builds a service call graph from metric data. Fault localization is based on the analysis of possible propagation paths over this graph, starting from the service node in which the anomaly is detected. If the analysis leads to multiple root cause candidates, these are ranked by their degree of association to the detected anomalous service, which MicroHECL measures by computing the Pearson correlation coefficient between observed metrics in the candidate and detected service nodes. MonitorRank and Microscope are similar in techniques with the exception of their heuristics to use metric data. MicroHECL was tested over 75 availability faults identified by operations engineers in the Alibaba e-commerce web-application, achieving a fault localization accuracy rate of 48%. For comparison, MonitorRank and Microscope achieved fault localization accuracy rates of 32% and 35% respectively when applied over the same data.

Trace-based approaches. We review as examples of this kind of approaches, Pinpoint [24], MEPL [17] and TFI [25], since they are significantly different from one another. Pinpoint is one of the early work that design and develops a trace-based fault localization technique. Unlike our approach that explicitly models fault propagation for each trace, Pinpoint learns and dynamically update a baseline model of the normal behavior of the application by aggregating tracing data generated by user requests. Pinpoint detects anomalies by comparing new requests against this model. For each detected anomaly, a decision tree analysis is carried out to locate the faulty component causing the anomaly. Pinpoint was tested on three applications of increasing complexity. In their experiments on Java applications, Pinpoint was able to detect just over 90% of the injected faults and successfully locate up to approximately 88% of the detected faults, resulting approximately in a 72% success rate for detecting and correctly locating faults.

MEPFL collects distributed traces from a target application running in a testing environment, where faults are injected in order to train a detection and fault localization model for use during production. This model consists of an input of expert-defined features populated from trace data and an output representing whether an error has occurred (detection) and, if so, which microservice (location) is causing it. MEPL achieves high precision and recall when it is evaluated on exactly the same faults in which the model was trained. However, performance drops significantly when tested on new faults (different than the ones used for training but of the same type), achieving precision between 58.6 - 98.4% and recall between 64.7 - 98.3% in detection, and a localization accuracy rate of 78.8% in localization.

TFI uses fault injection to populate a database of failures and corresponding trace fingerprints for a given application. TFI is triggered every time an issue is reported. The localization of the root cause is attained by finding the failure in the database whose trace fingerprint is the closest to the one collected when the issue occurred. This technique requires costly fault injection experiments for each application to build the database of fingerprints. The implementation of this approach by [17] only achieved a 11.7% fault localization accuracy rate on the same data in which MEPFL was tested on.

Putting FSF in perspective. Approaches that simulate faults in a testing environment to train a detection and localization model are limited by the type of faults considered during this phase and may lead to decreased accuracy in practice (e.g., as in the case of MEPFL). Also, these approaches assume that the application is fault-free before injecting any faults for training purposes. The existence of unknown faults in the application leads to noisy training data and models unable to recognize those faults in production. Supervised training also requires accurate labels, which may be difficult to achieve if we want to leverage production data, where it is not always possible to know what fault is in the system during a period of abnormal behavior: even if we see errors popping up, it might not be clear in practice what is causing those errors, so no fault location label can be specified. In addition, model training is time consuming and must be done for each new application. In particular, if the application is updated and changes are introduced, we will need to also update the model. FSF works directly on production data without the need to train any model. Thus, FSF does not have these limitations or the need to make these kind of assumptions.

In contrast, FSF focuses the computations only on traces with errors and use the application logs to find and convey causal explanations for the seen errors. None of these other approaches do that, or try any explainability beyond localization. Moreover, our proposed approach does not require training and is application-independent; making FSF ideal for adoption in production

VII. Discussion and Limitations

Limited fault models. In this work, we explicitly focus on service-level fail-stop-based faults (i.e., request failures). Thus, FSF, in its current form, cannot detect and localize certain class of faults (or requires additional modifications) as outlined:

(a) *Performance-related issues*. It is possible to convert observed anomalous tail-latency into a timeout fault by appropriately setting restrictive request timeouts [26]. FSF can potentially use model-based techniques to predict optimal timeout values [27], [28] or use SLO objectives to set timeouts; allowing FSF to detect and localize performance-related anomalies.

(b) *Infrastructure-related faults.* Infrastructure faults such as a host going down or network congestion can disrupt multiple services at the same time leading to multiple correlated service failures. Although FSF can detect and localize each of these service faults independently, it will not be able to identify the real cause which is in the underlying infrastructure. However, it is possible to combine infrastructure topology with traces to identify infrastructure faults uniquely as long as there is sufficient differential observability [29], [30].

Tracing overhead. Our approach assumes that distributed tracing is enabled for the supported application. This does not come without some work to make it happen. The generation of distributed traces requires the instrumentation of the application's source code or its lower-level libraries. Although, nowadays, there is support for automatically instrumenting microservices written in the most popular programming languages and frameworks, reducing significantly the effort needed in practice to set up this kind of monitoring. In addition, the use of any tracing system will inevitably add some small overhead, as it is typically the case with any kind of monitoring, including logs and metrics. On the other hand, our work has shown the debugging benefits of having a distributed tracing system monitoring our microservice application, which in our opinion clearly make up for the instrumentation effort and overhead coming from this kind of monitoring. The evaluation results reviewed in Section VI along with the ones obtained in Section IV, show how the performance of fault localization approaches depends on the kind of input data they consume, regardless of the specific algorithms used by each approach, improving as they move from basing their computations on logs to metrics to distributed traces. Therefore, justifying the adoption of distributed tracing despite its overhead.

VIII. Conclusion

We have proposed a fault localization approach for quickly detecting and locating microservice faults at runtime in production environments. This approach was able to automatically detect and accurately pinpoint the location of every simulated fault that we injected into a well-known microservice application benchmark, Train-Ticket. We also address the need of causal explanations that will help in repairing a detected fault. To do so, we relied upon application logs, which typically have the information necessary to figuring out why a fault happened. The problem with logs is that there might be too many of them since every microservice is constantly generating logs. This creates a huge haystack in which we must search and find the few logs that will explain the cause of a specific fault. We have used fault location and span time information to find the needle in this haystack of logs.

In comparison with other approaches, we do not rely in supervised learning to discover causal relations. Thus, we do not need a training phase in which to inject faults or the curation of any historical fault data to run our approach. These techniques are limited by the quality and scope of their training data as well as the generalization capabilities of their models to unseen faults. Indeed, we have noticed a clear deterioration in the performance of these methods when applied over faults that were not in the training set. Also, some of their reported performance in the literature suffered from overfitting, since they did not clearly separate the test and training sets. From an empirical point of view, we have seen how our fault localization algorithm outperformed causal graph based methods in their ability to detect and locate faults that take down a service.

References

[1] Gartner. (2021) Gartner says cloud will be the centerpiece of new digital experiences. [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2021-11-10-gartner-says-cloud-will-be-the-centerpiece-of-new-digital-experiences

[2] S. Newman, *Building Microservices*. O'Reilly Media, Inc, 2021.

[3] M. Waseem, P. Liang, and M. Shahin, "A systematic mapping study on microservices architecture in DevOps," *Journal of Systems and Software*, vol. 170, p. 110798, 2020.

[4] T. Betts. (2020) To microservices and back again – why segment went back to a monolith. [Online]. Available: https://www.infoq.com/news/2020/04/microservices-back-again/

[5] I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst, "Debugging distributed systems: Challenges and options for validation and debugging," *Queue*, vol. 14, no. 2, pp. 91–110, 2016.

[6] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 243–260, 2021.

[7] B. Li, X. Peng, Q. Xiang, H. Wang, T. Xie, J. Sun, and X. Liu, "Enjoy your observability: An industrial survey of microservice tracing and analysis," *Empirical Software Engineering*, vol. 27, no. 1, pp. 1–28, 2022.

[8] L. Wu, J. Tordsson, E. Elmroth, and O. Kao, "Causal inference techniques for microservice performance diagnosis: Evaluation and guiding recommendations," in *International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 2021, pp. 21–30.

[9] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou, "Sage: Practical and scalable ML-driven performance debugging in microservices," in *26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 135–151.

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

[11] D. Liu, C. He, X. Peng, F. Lin, C. Zhang, S. Gong, Z. Li, J. Ou, and Z. Wu, "MicroHECL: High-efficient root cause localization in large-scale microservice systems," in *43rd International Conference on Software Engineering: SEIP*. IEEE/ACM, 2021, pp. 338–347.

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

[13] M. Jiang, M. A. Munawar, T. Reidemeister, and P. A. Ward, "Dependency-aware fault diagnosis with metric-correlation models in enterprise software systems," in *International Conference on Network and Service Management*. IEEE, 2010, pp. 134–141.

[14] P. Aggarwal, A. Gupta, P. Mohapatra, S. Nagar, A. Mandal, Q. Wang, and A. Paradkar, "Localization of operational faults in cloud applications by mining causal dependencies in logs using golden signals," in *International Conference on Service-Oriented Computing*, 2020, pp. 137–149.

[15] L. Wu, J. Tordsson, E. Elmroth, and O. Kao, "MicroRCA: Root cause localization of performance issues in microservices," in *Network Operations and Management Symposium*. IEEE/IFIP, 2020, pp. 1–9.

[16] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer, "Failure diagnosis using decision trees," in *International Conference on Autonomic Computing, 2004. Proceedings.* IEEE, 2004, pp. 36–43.

[17] 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 *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 683–694.

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

[19] A. Parker, D. Spoonhower, J. Mace, B. Sigelman, and R. Isaacs, *Distributed tracing in practice: Instrumenting, analyzing, and debugging microservices*. O'Reilly Media, 2020.

[20] Y. Shkuro, *Mastering Distributed Tracing: Analyzing performance in microservices and complex systems*. Packt Publishing Ltd, 2019.

[21] A. Khatri and V. Khatri, *Mastering Service Mesh: Enhance, secure, and observe cloud-native applications with Istio, Linkerd, and Consul*. Packt Publishing Ltd, 2020.

[22] Train-Ticket, https://github.com/FudanSELab/train-ticket.

[23] L. Sun and D. Berg, *Istio Explained*. O'Reilly Media, Inc., 2020.

[24] E. Kiciman and A. Fox, "Detecting application-level failures in component-based Internet services," *IEEE transactions on neural networks*, vol. 16, no. 5, pp. 1027–1041, 2005.

[25] C. Pham, L. Wang, B. C. Tak, S. Baset, C. Tang, Z. Kalbarczyk, and R. K. Iyer, "Failure diagnosis for distributed systems using targeted fault injection," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 2, pp. 503–516, 2017.

[26] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An intelligent fine-grained resource management framework for SLO-oriented microservices," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 805–825.

[27] M. Allen, R. Wolski, and J. Plank, "Adaptive timeout discovery using the network weather service," in *11th IEEE International Symposium on High Performance Distributed Computing*, 2002, pp. 35–41.

[28] T. Dai, J. He, X. Gu, and S. Lu, "Understanding real-world timeout problems in cloud server systems," in *IEEE International Conference on Cloud Engineering*, 2018, pp. 1–11.

[29] P. Huang, C. Guo, J. R. Lorch, L. Zhou, and Y. Dang, "Capturing and enhancing in situ system observability for failure detection," in *13th USENIX Symposium on Operating Systems Design and Implementation*, 2018, pp. 1–16.

[30] S. Jha, S. Cui, S. S. Banerjee, T. Xu, J. Enos, M. Showerman, Z. T. Kalbarczyk, and R. K. Iyer, "Live forensics for hpc systems: A case study on distributed storage systems," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–16.