LongTale: Toward Automatic Performance Anomaly Explanation in Microservices

Richard Li
openrichard@fb.com
Meta
Menlo Park, CA, USA

Min Du
min.du.email@gmail.com
Palo Alto Networks
Santa Clara, CA, USA

Zheng Wang
gregdobbywz@gmail.com
University of Utah
Salt Lake City, UT, USA

Hyunseok Chang
hyunseok.chang@nokia-bell-labs.com
Nokia Bell Labs
New Providence, NJ, USA

Sarit Mukherjee
sarit.mukherjee@nokia-bell-labs.com
Nokia Bell Labs
New Providence, NJ, USA

Eric Eide
eeide@cs.utah.edu
University of Utah
Salt Lake City, UT, USA

ABSTRACT

Performance troubleshooting is notoriously difficult for distributed microservices-based applications. A typical root-cause diagnosis for performance anomaly by an analyst starts by narrowing down the scope of slow services, investigates into high-level performance metrics or available logs in the slow components, and finally drills down to an actual cause. This process can be long, tedious, and sometimes aimless due to the lack of domain knowledge and the sheer number of possible culprits. This paper introduces a new machine-learning-driven performance analysis system called LongTale that automates the troubleshooting process for latency-related performance anomalies to facilitate the root cause diagnosis and explanation. LongTale builds on existing application-layer tracing in two significant aspects. First, it stitches application-layer traces with corresponding system stack traces, which enables more informative root-cause analysis. Second, it utilizes a novel machine-learning-driven analysis that feeds on the combined data to automatically uncover the most likely contributing factor(s) for given performance slowdown. We demonstrate how LongTale can be utilized in different scenarios, including abnormal long-tail latency explanation and performance interference analysis.

CCS CONCEPTS

• **Networks** → **Cloud computing**; • **Computing methodologies** → *Machine learning*.

KEYWORDS

cross-layer tracing; performance analysis; tail latency

ACM Reference Format:
Richard Li, Min Du, Zheng Wang, Hyunseok Chang, Sarit Mukherjee, and Eric Eide. 2022. LongTale: Toward Automatic Performance Anomaly Explanation in Microservices. In *Proceedings of the 2022 ACM/SPEC International Conference on Performance Engineering (ICPE '22), April 9–13, 2022, Beijing, China.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3489525.3511675

ICPE '22, April 9–13, 2022, Beijing, China.

ACM ISBN 978-1-4503-9143-6/22/04…$15.00
https://doi.org/10.1145/3489525.3511675

1 INTRODUCTION

The microservices architecture [30] has been widely adopted for cloud-native applications, but its deployment in the wild brings many challenges. In particular, as the number of services and the complexity of their interdependencies grow within a given cloud application, their performance diagnosis becomes increasingly challenging. When a performance anomaly is encountered, the current best-practice approach is to use a combination of application-layer tracing and host-based performance profiling to drill down to its root cause. That is, the troubleshooting process starts with locating the bottleneck service using application-layer tracing, then queries high-level metrics (e.g., CPU/memory/IO utilization) on the host where the service instance is running to get a sense of which measures look suspicious, and finally drills down to potential culprits for the slowdown by inspecting system state revealed by performance profiling tools.

Let's consider *long tail latency analysis*. Figure 1 shows the latency distribution of a particular RPC, which is obtained from a group of five load-balanced service instances deployed on multiple hosts. While it is common to have tail latency in real-world distributed applications, many questions can be raised against this simple plot. For one, is the tail in this CDF normal? Even this simple question is hard to answer because it is typically infeasible to define the "normal" baseline behavior of a given microservice when it goes through the rapid development life cycle of continuous integration and continuous delivery (CI/CD). Even if we manage to conclude that the tail latency deviates from its (projected) normal behavior, how can we know which service instance out of five contributes the most to the tail? Is it only one service instance contributing to the slowdown or are all the instances to blame for the degraded performance? Is the culprit long-lasting performance interference or short-lived microbursts in resource usage? In the latter case, how can we detect the presence of microbursts if they are randomly scattered around the whole fleet of service instances?

Conducting analysis to answer these questions is challenging for the following four reasons.

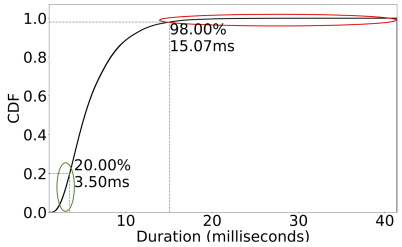


Figure 1: Example CDF of RPC latency.

- **Horizontal complexity.** Real-world microservice deployments easily include hundreds of different types of microservices [1, 13, 31], and each service independently scales out based on load. Such massive-scale, distributed deployments pose a major hurdle for narrowing down the scope of investigation. Imagine a performance degradation case where slow RPC transactions happen only sporadically, and the set of bottleneck services and physical servers involved varies from time to time. In this case, it is unclear which service or physical server an analyst should start investigating.
- **Vertical complexity.** Even after an analyst manages to zero in on the candidate services or physical servers to investigate, there are too many performance metrics and subsystems to inspect across the entire software stack and operating system. Thus it becomes another challenge to decide on the right set of metrics and establish causality between the chosen metrics and a given performance anomaly.
- **Operational knowledge.** After having identified key contributing metric(s) to help explaining a given performance anomaly, an analyst often resorts to advanced performance profiling and hotspot analysis tools to ultimately pinpoint the application process or a kernel component that leads to such slowdown. Wielding those tools itself is a non-trivial work.
- **Domain knowledge.** An analyst is typically not familiar with the codebases of all the services. In the microservices architecture, every development team takes ownership of its functionality with proper APIs exposed to other teams, and its service is treated as a black box by other teams. Without any domain knowledge, it is challenging for an analyst to corroborate his or her findings.

Faced with these formidable challenges, we step back and start with a very simple, intuitive question: *what if we could simply compare the slow and fast responses from a microservices-based application to understand what is going on when a response is slow?* More specifically, if we can somehow compare the snapshots of individual microservices' execution contexts captured at the head and the tail of the CDF curve shown in Figure 1, can we explain the slowdown? In a nutshell, this is the question we set out to explore. The natural next question is whether or not existing performance diagnosis approaches can answer this question effectively. The answer is no, as we clarify in the following.

There are mainly two families of tracing tools that have been utilized for performance troubleshooting. Distributed application-layer tracing solutions [4, 26, 28, 35] have been proposed in industry and academia. While useful for identifying service-level dependencies and detecting performance anomalies at service granularity, these solutions are inadequate for diagnosing and explaining detected anomalies, mainly because of lack of detailed execution contexts associated with application-layer traces. Besides, since application layer tracing only collects events that are on the execution path of applications, it does not help uncover the root causes that happen off the execution path [43]. To complement application-level tracing, the system-level profiling tools [2, 6–9, 17, 21] provide rich information about the running state of the host operating system and its workloads. However, the complexity of the current operating system necessitates that a diagnostician pose numerous questions and hypotheses about potential root causes in order to proceed step by step towards a reasonable explanation. Making the right hypotheses and picking the right tools to start with are non-trivial tasks even for an experienced diagnostician. For each subsystem, an analyst typically utilizes a different set of tools [23] for deeper level inspection to figure out the root cause of the anomaly, which requires substantial domain-specific knowledge and familiarity with the tools.

In order to address these shortcomings of the existing approaches, we propose a new approach called LongTale that combines the advantages of application-layer tracing and host-based performance profiling, with the ultimate goal of enabling *fully automatic performance anomaly explanation*. LongTale achieves this though cross-layer trace stitching and machine learning-driven anomaly explanation. More specifically, LongTale stitches application-layer traces with corresponding system stack traces. This cross-layer trace stitching allows the huge amount of runtime contexts of microservices to be properly sliced and reorganized to focus only on the system states that are associated with two opposite regimes (i.e., head/tail) of latency distribution. LongTale feeds this reorganized dataset into machine learning algorithms for regression and feature selection, which ultimately highlights the most likely contributing factor for given performance slowdown.

LongTale advances the existing state of the art tracing and diagnostic systems in three aspects.

- **Non-intrusiveness:** It is designed to avoid any application-specific adaptation or kernel level change by relying solely on data-oriented analysis. Such transparency allows it to be immediate deployable.
- **Ease of use:** It requires no knowledge about microservices to be deployed and the underlining infrastructure.
- **Automated root cause diagnosis:** Instead of focusing on suspicious high-level metrics, which are just an artifact of a real root cause, LongTale is able to pinpoint the root cause itself without an analyst going through an arduous procedure of repeated hypothesis testing. This capability is especially useful in diagnosing microburst-induced anomalies which cannot easily be detected with coarse-grained resource usage monitoring.

We evaluate LongTale by injecting simulated anomalies in a real-world microservices-based e-commerce application. Experimental results show that LongTale can help an analyst better understand the behavior of the anomalies by automatically and precisely pinpointing culprit processes with low false positive rate.

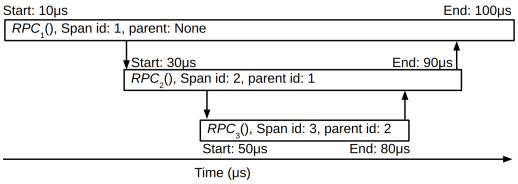
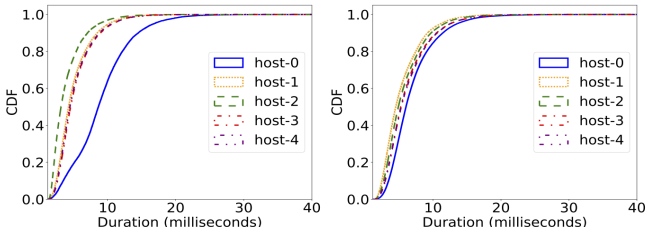


Figure 2: Trace and Spans in application-level tracing.



(a) Single-host anomaly **(b) Multi-host anomaly**

Figure 3: Effect of anomalies on latency distribution.

The rest of the paper is structured as follows. After covering a brief overview of existing tracing tools in Section 2 and a motivating scenario in Section 3, we present the LongTale architecture in Section 4. Next, we describe the LongTale prototype implementation in Section 5 and its detailed evaluation in Section 6. After a review of related works in Section 7, we discuss the scope of our work in Section 8 and conclude in Section 9.

2 BACKGROUND

Before describing LongTale, we provide a brief overview of tracing technologies that it builds on.

Application-layer tracing. Application-layer tracing collects timestamped application events in the form of *Traces* and *Spans* via instrumented RPC/web frameworks [36]. A Trace represents a series of operations executed by an application due to an external event (e.g., a user sending a request to an application). A Span represents a particular action, such as an RPC call or a function call. A Trace is constructed as a directed acyclic graph (DAG) of one or more Spans. Each Span logs an operational context including operation name, start/end timestamps, and its relationship with other Spans. An edge between a Span and its parent Span indicates a causal relationship between them. Figure 2 shows an example Trace composed of three Spans, where RPC_3 is invoked by RPC_2, which in turn is triggered by RPC_1.

Stack trace monitoring. A stack trace of a process captures active stack frames for that process at a particular point in time. From the stack trace, one can extract the detailed execution contexts of the process during that time (i.e., chains of function calls that are activated). In reality, the existing system-level profiling tools [8, 17] that can record stack traces produce call chains of memory addresses ($addr_A \rightarrow addr_B \rightarrow addr_C$, etc.). Thus, performance diagnosis with stack traces requires an additional symbol resolution step that translates the raw stack traces into symbolic call chains ($func_A \rightarrow func_B \rightarrow func_C$). Using these existing tools, one can inspect system-wide runtime contexts of a server by taking a global snapshot of stack traces for *all* active processes including kernel worker threads. Taking one step further, one can collect such system-wide stack trace snapshots periodically and aggregate them over time to obtain approximated fine-grained CPU usage attribution for different functional components of active processes (e.g., flame graphs [21]).

3 MOTIVATING SCENARIO

As a motivating scenario we consider two possible variations of the abnormal long tail latency depicted in Figure 3. In this scenario, five replica service instances are deployed across five different hosts, with a load balancer evenly distributing the load among them.

In one abnormal case (Figure 3a), a long-lasting computational workload is artificially injected on host 0, affecting the latency of its service instance, while the other hosts remain normal. Such long-lasting performance interference anomaly can easily be detected from the CDF plot. An analyst can then collect information at the problematic host to determine which process is introducing the slowdown. The analyst can use "baseline" data to compare different execution states of the "normal" and "abnormal" hosts. In fact, this type of comparative performance regression has become a common practice in companies such as Netflix [22].

In the other abnormal case (Figure 3b), all five hosts experience multiple scattered microbursts in CPU usage from a series of short-lived performance interference events. While all five service instances handle the workload with almost the same latency, these microbursts cause abnormal long-tail latency. In this case, an analyst may find it difficult to conduct the investigation for the following reasons.

First, since each host performs similarly, it is difficult to decide which host an analyst should start with. Second, coarse-grained resource usage monitoring may not be helpful for pinpointing a period of execution worthy of investigation, especially if the duration of a microburst is shorter than the interval of periodic metric collection. For example, the `top` command's default refresh interval is 3 seconds. Any CPU saturation event that does not outlast this interval will not be reported as such. Even if CPU saturation does get detected via performance monitoring tools, the spike cannot easily be correlated with long service latency. After all, resource saturation is not an uncommon event, and most spikes do not lead to a slowdown. Third, inspecting system-wide stack traces like Flame Graphs on individual hosts does not help much, because the microbursts may be spread out so sparsely that they only take a minuscule amount of CPU quota, which can easily be neglected by an analyst. Fourth, application-layer tracing alone can barely help. Application-layer traces can highlight the Spans that take much longer than the majority. However, without any runtime contexts behind the slow Spans, an analyst can neither draw any conclusion nor proceed further with the investigation.

It is worth noting that microburst-induced anomalies can also prevent autoscaler from properly relieving the slowdown. An autoscaler is supposed to spawn new replica services when the resource saturation level exceeds a preset threshold. In reality, the replica spawning operation comes with a delay in order not to waste

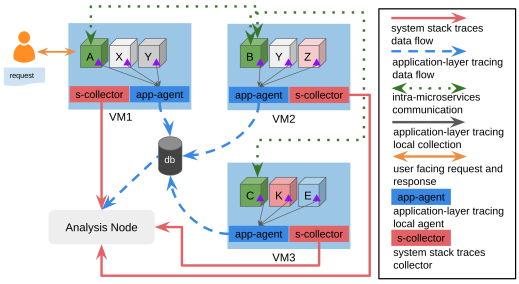


Figure 4: LongTale architecture.

resource by responding too quickly to short-lived resource spikes. For example, the autoscaler of Kubernetes uses 15 seconds as its default control period [40]. Therefore, only resource saturation that outlasts this period will cause the autoscaler to spawn new replicas. This means that autoscaling will never be activated by microbursts as their duration is typically much shorter than 15 seconds.

In short, the abnormal long-tail latency resulting from scattered performance interference can be extremely difficult to explain, not to mention relieving it. This becomes a needle-in-a-haystack problem. LongTale was initially designed to tackle this kind of anomaly, but it is useful for diagnosing other kinds of performance anomalies as well. In this paper, we apply LongTale to a sparse microburst-induced anomaly as well as several other anomalies.

4 SYSTEM DESIGN

In this section, we articulate the design goals driven by the motivating scenario. Based on the design goals, we sketch the LongTale architecture and detail the end-to-end workflow.

4.1 Design Goals

In order to address the challenges posed by the motivating scenario, we set four design goals for LongTale.

Application agnostic. To be a viable performance diagnosis tool for microservices, the first goal is *generality across different microservice implementation.* A typical cloud application consists of a multitude of distinct microservices which are highly heterogeneous in terms of programming languages and development platforms used. It is unrealistic to expect any diagnostician to have any in-depth knowledge about the microservices under examination.

Immediately deployable. Expecting production systems to adapt their software stack to a debugging tool is impractical. This is true especially for small- to mid-size organizations that rely on publicly available open-source software, but that lack the capability or manpower to retrofit the deployed software and available tracing libraries for more advanced performance diagnosis. Besides, there is always a risk that any adaptation for debugging may lead to software crashes if not thoroughly tested. Any troubleshooting solution that *requires minimal adaptation for microservices and the underlying operating system* can be immediately deployable.

Unobtrusive. The performance overhead of a debugging or monitoring tool is an important factor to consider. If the tool collects information too aggressively, it may directly lead to performance regression of deployed services, either lower throughput or higher latency. It is also possible that the overhead introduced by the tool may actually prevent an existing bug from showing up, which is known as "Heisenberg effect" of the tracing system. To avoid both of these problems, the tool needs to only rely on *unobtrusive measures.*

Informative. The tool should *point an analyst directly to the root cause*, e.g., a culprit process, instead of flooding the analyst with too much metric data or too huge a hypothesis space to explore. Besides automated end-to-end explanation, the tool should provide suitable visual means for the analyst to understand the running state of the distributed system.

4.2 Architecture Overview

LongTale works by combining an application-layer tracing framework with host-based tracing. Figure 4 presents its architecture overview. The figure show that microservices are deployed across three VMs. Each service runs in a container instantiated in any of these VMs. Users issue requests and receive responses through a front-end service or a load balancer. To service a user's request, multiple services may be triggered to execute specific functions to complete the task. In the figure, the service A, B, and C are collaborating to complete the task. Transparent to the deployed microservices, there are two subsystems in LongTale that collect information for anomaly detection and diagnosis. One is the centralized application-layer tracing subsystem, and the other is the host-based tracing subsystem which is operated on each host.

In the application-layer tracing subsystem, each container—i.e., service—has a built-in application-layer tracing component that records the functions triggered by a request along with timestamps. The tracing component accumulates the data and send it to the application-layer tracing agent (the *app-agent* in the figure) that is running on each host. In the host-based tracing subsystem, the host-based agent (the *s-collector* in the figure) collects information that is complementary to the data gathered by application-layer tracing. Unlike application-layer tracing, it collects information that is pertinent to the local host where it is running. This information is not just about local containers, but system-wide traces for all active processes including kernel threads on the host. The data collected by the app-agent and the s-collector will both be transferred to the analysis node for machine-learning-based diagnosis.

4.3 End-to-end Workflow

LongTale combines the information from the two tracing subsystems into a structure that is well suited for machine learning analysis. Figure 5 shows the end-to-end workflow of LongTale. Data from application-layer tracing and host-based tracing goes through a multi-stage pipeline to produce a diagnosis result.

4.3.1 Trace Sanitizer. The information collected by a host-based tracing tool (e.g., `perf`) is system-wide stack traces for a given host. These stack traces require additional preprocessing before being fed into the LongTale's pipeline. First, the original stack traces which consist of call chains in the form of memory addresses need to be resolved into symbolic call chains (e.g., $func_A \rightarrow func_B \rightarrow func_C$) for subsequent stages of the pipeline. Symbol resolution

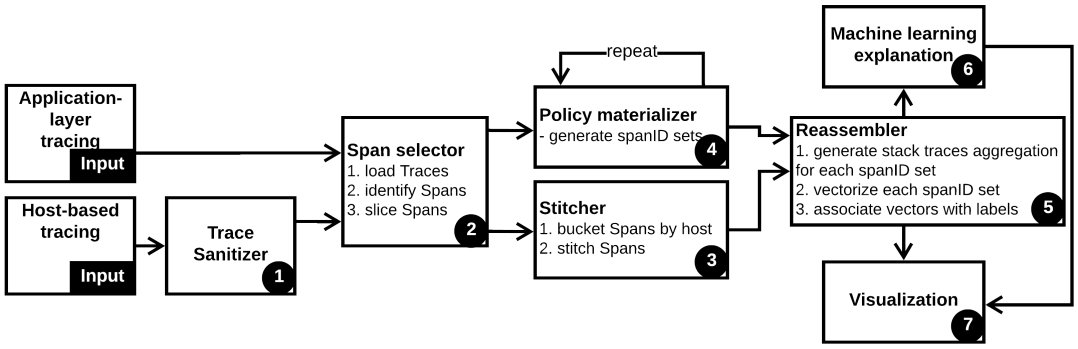


Figure 5: LongTale end-to-end workflow.

can be performed on the host where the traces are collected or on a dedicated machine.

One issue is that periodically sampled stack traces may be subject to *sampling bias* due to Dynamic Voltage and Frequency Scaling (DVFS) of modern CPU processors. That is, even with a preconfigured fixed sampling rate of host-based tracing, the actual sampling rate can dynamically fluctuate depending on the load of the host [38], which will introduce a bias in the subsequent Reassembler stage (see Section 4.3.5). To tackle this problem, we regularize collected stack traces by inserting *virtual* samples. Specifically, each time a stack trace sample is collected, the tracing tool also records its duration. If a given sample spans across multiple sampling intervals, we insert as many duplicates of this sample (virtual samples) to cover its entire duration. The sampling interval in host-based tracing is set based on the maximum CPU clock frequency.

4.3.2 Span Selector. LongTale allows an analyst to narrow down his or her investigation to a particular Span. This is not possible in existing application-layer tracing systems as they allow users to monitor services only on a Trace granularity. To achieve Span-level filtering, Span Selector pulls all collected Traces for a given service, and walks each Trace to pick only those Traces that contain the Span in question. After all such Traces are retrieved, Span Selector slices the Span in each Trace, such that only the time when the Span is active within the service is recorded. Take Figure 2 for example. If the target Span is RPC_2, then Span Selector will cut out the duration of RPC_3 from the target Span, so that we only account for the portion of RPC_2 that does not include the time when the Trace has gone off to another service. After this slicing, the remaining Span of RPC_2 will have two separate durations 30μs—50μs and 80μs—90μs. For a Span that is the innermost in a Trace, we do not need to slice it.

4.3.3 Stitcher. Once sliced Spans and stack traces are ready, the LongTale's pipeline moves on to stitching, where they are combined. Stitching occurs on a per-Span basis as follows. For each Span created on a particular host, Stitcher pulls the corresponding stack traces that are collected from the same host and that fall within the Span's execution time window. By stitching the Spans and stack traces collected from the same host, we can avoid *timing skew* that is inherent in any distributed system. Both Spans and stack traces are recorded with timestamps from the same monotonic clock. By combining the Spans and stack traces collected in exactly the same time window, we aim to explain a given Span's application-layer behavior (e.g., latency) from the corresponding microservice's complete runtime contexts.

4.3.4 Policy Materializer. As a general-purpose performance diagnosis tool, LongTale exposes a flexible policy interface through which an analyst queries the system for anomaly diagnosis. The role of Policy Materializer is to materialize a user-submitted policy into the corresponding trace data set that will be fed into machine learning analysis. An example policy is to compare Spans shorter than 10%-percentile with those longer than 98%-percentile for a particular service.

Our experience with real-world trace data is that if we apply our machine learning algorithms to individual Span instances, they often fail to yield a valid result especially when Spans are active for a very short period of time (e.g., on order of milliseconds). This is because each short-lived Span only has few stack trace samples associated with it, from which no significant runtime context can be learned. Only when we accumulate an enough amount of stack trace samples across time, can the machine learning algorithms learn the statistical characteristics of host-wide CPU allocation, and properly infer a given Span's runtime context.

Drawing on this observation, Policy Materializer, in response to a user-submitted policy, instantiates a set of randomly chosen Span instances that satisfy the policy (denoted as *spanID set*). It ensures that the number of instances in the spanID set is enough that the corresponding stack trace samples in aggregate cover at least 10 second duration. Policy Materializer repeats this step to generate M different spanID sets for machine learning analysis. In case of the above example policy, Policy Materializer prepares a pair of M spanID sets, one for 10%-percentile and the other for 98% percentile. LongTale supports other policies for different types of diagnoses. We detail them further in Section 6.

4.3.5 Reassembler. The role of Reassembler is to convert the spanID sets prepared by Policy Materializer into the vector-formatted feature sets to be consumed by the machine learning analysis. Specifically, for each spanID set, it puts together all the stack trace samples that are associated with the set and counts the number of times

Table 1: Feature set example generated by Reassembly.

Vectors	Feature 1	Feature 2	...	Label
vector-1	0.23	0.31	...	8.5ms
vector-2	0.33	0.21	...	9.7ms
vector-3	0.24	0.29	...	9.1ms
...	...	...	...	...

each function appears at a particular depth of the stack traces. After counting, it normalizes the counts and turns the data into a vector, e.g., [$func_A$: 0.23, $func_B$: 0.31, ...]. With M spanID sets, Reassembler generates M corresponding vectors.

One design decision during this conversion step is *which depth* of stack traces should be used as features. The deeper a given stack is, the more functions will be available as features to select and learn. Too many features may lead to the data sets being too noisy and the machine learning models becoming less accurate on prediction and explanation. An analyst can either choose the bottommost stack frames or focus on those at a specific depth. Either choice has its own benefit. Utilizing the bottommost stack frames summarizes process-level CPU allocation, while deeper layers into the stack can capture relative CPU allocations of particular functions within a process.

LongTale provides an interface for an analyst to explore both scenarios. By default, LongTale will pick the bottommost stack frames to pinpoint which processes are the culprit for a given performance slowdown. By doing that, LongTale excludes many noisy features so that the culprit can stand out from the features clearly. If an analyst needs to dive deeper to identify which functional module or subsystem of a chosen process contributes the most to the slowdown, he or she can pick a depth into the stacks to repeat the analysis.

Reassembler labels each vector with the average duration of all the Spans belonging to the original spanID set. This label will be used for linear regression and feature selection in the later stage. The final output generated by Reassembler is in the form of a matrix of numbers, as shown in Table 1.

4.3.6 Explanation Using Machine Learning. The ultimate goal of LongTale is to find out the most contributing factor(s) for a given performance slowdown. Achieving this goal really boils down to identifying the most distinguishing feature(s) between the two vector sets generated from the previous stage. This problem can be formulated as a *feature selection problem* in predictive modeling, which can be solved by traditional machine learning algorithms.

LongTale uses ElasticNet to perform the feature selection task. As a regularization regression method, ElasticNet combines the properties of both Ridge Regression and LASSO regression so that it can make an automatic variable selection and continuous shrinkage while preserving the sparsity of representation [46]. Feature selection aims to filter out irrelevant or redundant features, resulting in a trimmed set of relevant features. ElasticNet performs feature selection automatically as part of learning the model, which is also known as intrinsic feature selection method. As a linear regression model, the output of ElasticNet model training contains the coefficient of each feature, of which the sign represents how the feature is correlated to the target variable, positively or negatively, and the scale the strength of the correlation.

In this stage, we feed the ElasticNet algorithm with the generated vector sets and labels for the model to learn the coefficients of each feature. If an anomaly introduces any discrepancy in the vector sets, the learned coefficients associated with the elements in the vector will be sparse, i.e., most coefficients are zero or close to zero, and the coefficients of the culprit can be the largest. On the other hand, if there is no performance anomaly, and the long tail latency simply originates from natural queuing delay, the algorithm will not find any dominant feature with a coefficient that is positively correlated to the Span duration. We consider those coefficients with low standard deviation as anomaly-free, and those with high standard deviation anomalous. In LongTale, we use the 99% confidence interval as a threshold metric for identifying culprits.

4.3.7 Visualization. Effective visualization is important for diagnosing large-scale distributed systems [21]. LongTale provides histogram-based diagrams that represent the importance of features in causing a given performance anomaly, a heatmap-based visualization to demonstrate when and where the anomalies have occurred across different time periods and hosts (Figure 7), as well as aggregated flame graphs for comparative analysis (Section 6.5.2).

5 IMPLEMENTATION

As a proof of concept, we develop a prototype of LongTale. In the following, we highlight a few important details of the prototype.

We deploy Jaeger [3] (v1.16.0) for application-layer tracing, and utilize the `perf` tool (v4.15.18) for stack trace collection. While application-layer tracing is enabled all the time, host-based tracing is activated on demand when a user submits a policy for troubleshooting (e.g., with a particular service/Span). Upon policy submission, the s-collector running on the hosts where the target service/Span is active will invoke `perf` for stack trace collection. By default, stack traces are collected for 10 minutes. Once the default round of stack trace collection is completed, stack traces are sanitized locally on individual hosts, and uploaded to the analysis node, which will trigger the rest of LongTale's workflow pipeline.

Span selection is performed on the analysis node by using the existing Jaeger query API against the Traces and Spans collected by the app-agent (realized with Jaeger agent). Due to the API's limit on the amount of Traces being retrieved each time, the Traces are retrieved in smaller chunks. Once Traces are ready, Span slicing and stitching steps get started. While stitching is performed once on the available data, the stitched data needs to be accessed many times for the subsequent machine learning analysis (for generating multiple random instances of spanID sets). Thus, the stitched traces are stored in a separate key-value store, where the key is each Span's unique ID and the value is the information associated with the Span, including its application-layer metadata and combined stack trace samples.

An alternative implementation is to offload the stitching operation to individual hosts. That is, the Jaeger agent could be modified to perform slicing and stitching on collected trace data and push the processed result to the analysis node. While that would make trace processing scale naturally with the growing number of microservices, it could introduce unduly significant performance penalty to

the deployed microservices. Besides, such approach would restrict the type of stitching operations performed by LongTale. Decoupling trace collection (done on end hosts) and trace stitching (performed on the analysis node) makes stitching much more flexible and allows LongTale to be easily extended to incorporate different type of system metrics beyond stack traces when troubleshooting production systems.

We use `sklearn` [5] (v0.23.1) with default parameterization to implement ElasticNet-based learning and feature selection.

Our prototype consists of 2,800 lines of Python code for the entire LongTale processing pipeline, and 1,600 lines of Shell code for host-based stack trace collection and sanitization. We plan to release the source code of the prototype in the near future.

6 EVALUATION

We evaluate the LongTale prototype by testing its ability to diagnose performance anomalies. In this evaluation, we focus on answering three questions. *First, can LongTale automatically explain the abnormal tail latency in microservices, which is caused by a performance interference event (§6.2)?* According to our experiment, the answer is yes: LongTale accurately pinpoints the root cause for the performance slowdown through stitching and learning on the collected trace data. We also demonstrate what the explanation result would look like when there is no anomaly to be detected, as well as when there are multiple anomaly factors involved in the slowdown. We show how LongTale can visualize analysis results to help an analyst understand when and where anomalies are happening in a given application deployment. *Second, is LongTale general enough to work for different types of microservices developed in different languages without any domain-specific knowledge (§6.3)?* The answer is yes: LongTale works with every service we experimented with that is part of a real-world e-commerce application. These services are developed in Java, Node.js and Go. *Third, what is the performance of the LongTale's processing pipeline (§6.4)?* We provide a breakdown of completion time of its pipeline processing including data collection, preprocessing, and learning. Our experiments show that LongTale can complete an explanation task within reasonable time. Finally, we demonstrate two other use cases of LongTale to show its versatility (§6.5).

6.1 Experimental Setup

We deploy the LongTale prototype and run our experiments on a testbed consisting of six Dell PowerEdge R430 servers, each with two 2.4 GHz Intel Xeon E5-2630v3 8-core CPUs and 64 GB RAM. The testbed is provisioned within CloudLab [32].

On this testbed, we set up a cloud platform using `OpenStack` (Stein release) for VM provisioning, and `Kubernetes` (v1.18.3) for microservice creation and orchestration within VMs. A total of 22 VMs are instantiated by OpenStack on the testbed. Among them, one VM is used as a Kubernetes controller node, another as the Jaeger trace collector node, and the remaining 20 are used as compute nodes to deploy container-based microservices. The VMs used as compute nodes are each provisioned with one vCPU and 2GB RAM. All physical machines and VMs are running Ubuntu 18.04. We turn off turbo boost mode on every physical machine. We also set the CPUFreq "governor" as "performance" to make the CPU run at its rated frequency 2.4 GHz.

Table 2: Microburst configurations.

Config	Microburst description	Collection	STD
config-1	No microburst injected	10 min	4.69
config-2	5-sec duration every 1 min	10 min	166.14
config-3	3-sec duration every 1 min	10 min	61.21
config-4	5-sec duration every 3 min	20 min	63.55
config-5	3-sec duration every 3 min	20 min	27.48
config-6	5-sec duration every 5 min	30 min	52.89
config-7	3-sec duration every 5 min	30 min	138.12
config-8	Random spikes every 1-5 min with 3-5 sec duration	30 min	66.37

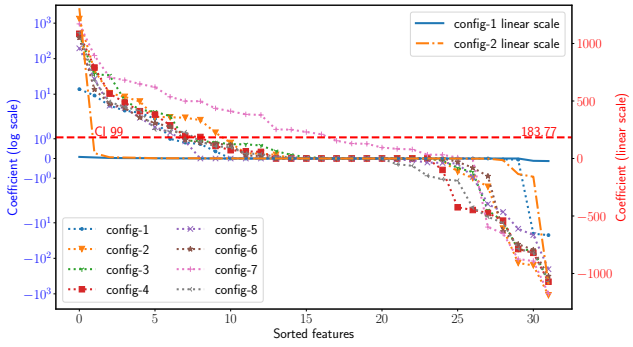


Figure 6: ElasticNet's feature selection results.

On top of a Kubernetes cluster, we deploy an e-commerce application called Sock Shop [39], which is composed of 14 different types of microservices developed in multiple languages. As a default configuration, six services among them (user, catalogue, carts, shipping, payment and orders) have application-layer tracing enabled. We scale up each service to five replica instances. We use `locust` (v1.0.2) as a workload generator for Sock Shop.

6.2 Anomaly Explanation

It is almost impossible to exactly reproduce documented real-world performance anomalies in our testbed environment due to the significant differences in microservice implementation and infrastructure settings. Instead, we inject artificial performance anomalies with a similar net effect.

First, we re-create the motivating scenario described in Section 3, where abnormal long tail latency is induced by microbursts of CPU saturation. Here we simulate the microbursts using `stress-ng`, a tool designed to stress a system in highly configurable fashions. We choose the `user` service that handles user account logins in Sock Shop as a target service, and explore the LongTale's capability to detect and explain the long-tail latency exhibited by the service when the intensity and duration of the injected microbursts are varied. For the experiment, we generate a synthetic workload for the service, that comprises 5,000 requests per second.

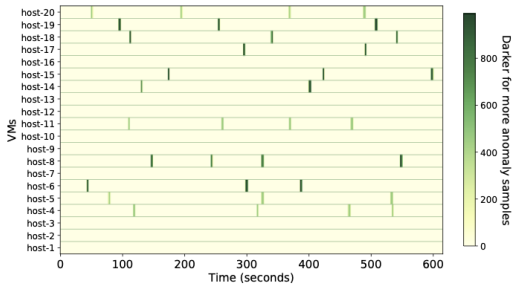


Figure 7: Heat-map visualization for scattered performance anomalies.

Table 2 shows eight different configurations of injected microbursts as well as trace collection time. Config-1 is the base line normal case without any anomaly. Figure 6 shows the ElasticNet's feature selection results for each of these configurations. The **STD** column shows the standard deviation for the feature coefficients computed by ElasticNet. We can see that the anomaly-free case (config-1) has a much lower standard deviation than any other configurations, i.e., its curve is almost flat in linear scale, shown as the blue solid line. We perform ten experiments in the anomaly-free case to compute their standard deviations. The average standard deviation among the ten cases is 4.69 with maximum of 7.23. We pick the standard deviation of ten as our threshold to decide whether an explanation result contains anomalies. We later show that the standard deviation of 10 works mostly well with different types of services with reasonably low false positive rate.

In Figure 6, features are sorted on the x-axis in an increasing order of their coefficients. We exclude all the features with zero coefficient, which correspond to 100 or so background processes running on the host. The ElasticNet algorithm considers these zero-coefficient features neither positive nor negatively correlated with the Span duration. A positive coefficient means positively correlated with the latency, and vice versa for negative coefficients. For example, in all these experiments, the most negative coefficient is associated with the `user` task process that serves the user login requests. This is because the more CPU quota is allocated to this process, the lower latency it will exhibit. Across all the anomaly cases (from config-2 to config-8), the `stress-ng` task process is assigned the largest positive coefficient.

To highlight different feature selections among seven different microburst scenarios, we use two different y-axes, plotting the results in log scale in the y-axis on the left-hand side, and in linear scale in the y-axis on the right-hand side. We can see that, when there is any dominating coefficient, the decline of the line is much sharper than the case without it which is almost flat. The horizontal red line marks the 99% upper bound confidence interval of config-2, which separates the outlier component from the other features clearly. LongTale explains correctly even when the microbursts happen only 1% of the time (config-7) and also the case when they are randomly scattered (config-8).

To help an analyst better understand when and where the microbursts are occurring, LongTale, based on ElasticNet feature selection, automatically generates a heatmap-based visualization as shown in Figure 7. In this example, host-2, 9, 12, 13 and 19 are the ones that run the `user` services. Our microburst injection script randomly selects a half of the 20 VMs, and in each of the chosen VM, injects randomized microbursts according to config-8. All these randomized microburst behaviors and their severity are clearly visualized from the plot. We can see that not all `user` service instances are affected by the microbursts (e.g., host-2 did not experience any anomaly). The identified anomalous processes also appear on the VMs that do not run the `user` services, which means that these microbursts may affect some of other services and their tail latency as well.

Actions can be taken after understanding why, when, and where the anomaly happens. System administrators can have different ways to act on this knowledge to relieve the abnormal long tail. Most microburst-induced anomalies are benign although they slow down some of the requests. Simply killing them is not a practical option especially if they are unavoidable (e.g., maintenance tasks). One possible way is to make the Kubernetes scheduler aware of a potential slow down on a specific instance, so that it can avoid redirecting traffic to the instance. Since microservices instantiated by Kubernetes typically run behind its built-in service discovery and load balancing mechanism, the load balancer can be easily notified to dynamically adjust load distribution. This is in fact commonly adopted best practice for cloud infrastructures to improve their availability and latency [12]. Similarly, we can automatically configure the Kubernetes load balancer to deactivate load distribution on the instance before the execution of an anomalous process (e.g., off-path maintenance events), thereby avoiding the long latency induced by the anomaly. When the anomalous process completes its work, we can lift the pause action on the instance. We leave detailed exploration on automatic mediation on abnormal long tail latency as future work.

6.3 Service-Agnostic Anomaly Diagnosis

A performance diagnosis tool should be general enough to work with different types of services implemented in different languages. To demonstrate the service-agnostic diagnosis capability of LongTale, we now perform experiments with six other services in Sock Shop. Three of these services are developed in Go, and the others in Java. We inject microburst events to each of these service with the severity of config-2 in Table 2. In each service, we repeat feature selection experiments 10 times to compute an average standard deviation for ElasticNet coefficients. We also obtain false positive rate (FP%) and false negative rate (FN%) of anomaly detection when the 99% upper bound confidence interval is used as a threshold. The results are summarized in Table 3.

According to the table, all tested services have zero false negative rate except for the `payment` service. The high false negative rate of `payment` is linked to its relatively low standard deviation. The RPC of the `payment` service is implemented as a simple stub function that merely echoes whatever is sent by the client. Such simple design of the service is so lightweight that the injected CPU microburst events can barely affect its latency performance. As a result, the behavior of the service, even with microbursts, is almost identical to that of an anomaly-free deployment, and the ElasticNet algorithm does not produce dominant-enough coefficients that

Table 3: Anomaly explanation results across different services.

Service	Lang.	Span	STD	FP%	FN%
carts	Java	get-items	785.5	0.28%	0
shipping	Java	post-shipping	1553.9	0.39%	0
order	Java	get-collection-resource	830.1	0.19%	0
user	Go	login	136.4	0.0%	0
catalogue	Go	GET /catalogue/{id}	156.6	0.49%	0
payment	Go	POST /paymentAuth	12.2	0.34%	60%

Table 4: LongTale pipeline performance breakdown.

Stage	Process time
Trace Sanitizer	4.13 min
Span Selector	37.55 min
· Trace loading	4.18 min
· Span loading/filtering	33.37 min
Stitcher	1.69 min
Vector Generation (20K)	25.69 min
Explanation	2.83 sec
Total	69.11 min

capture the anomaly. This result is in fact in line with the observation made in MicroRCA [41]. Nevertheless, LongTale's explanation result still considers `stress-ng` as one of the most positively correlated features, and its coefficient always ranks among the two biggest coefficients.

The false positive rate is reasonably low across all the services. Another metric (not shown in the table) is the precision, which is the number of true positives divided by the number of true positives plus the number of false positives. This metric is important as it gauges the amount of extra effort required for anomaly investigation. Across all the services, we observe the precision above 80–90%, except for the `payment` service (25%). In terms of the absolute number of false-positive detections, there have never been more than two false positives across all services including `payment`. This means that LongTale does help an analyst narrow the candidate set of more than a hundred active processes to investigate down to just two or three suspicious processes, which significantly shrinks the hypothesis space that needs to be explored.

6.4 Diagnosis Performance Breakdown

Table 4 shows the breakdown of completion time for the diagnosis of a microburst-induced performance anomaly (config-2). We can see that the time required for machine learning explanation is very short (order of seconds). Note that the rather slow completion time of other steps is an artifact of our prototype implementation (e.g., single-threaded chunk-based download of Trace data via Jaeger APIs). Each of these steps can easily be expedited with multi-threaded processing. It is also important to note that, even with a large-scale real-world cloud application built with a large number of microservices, typical Traces/Spans traverse *only a limited number of service instances*. Thus, we do not expect the LongTale's pipeline execution time to grow significantly.

Performance overhead added by LongTale for deployed microservices comes from both application-layer tracing and host-based stack tracing. These types of tracing have been designed to introduce acceptable performance overhead with a reasonable sampling strategy. Application-layer tracing has already been proven to be unobtrusive in production systems [35], and even the overhead of instrumenting every single request is considered modest [43]. Similarly for stack tracing by `perf`, we use an acceptable sampling rate of 999Hz, which is widely used in industrial production systems [24].

6.5 Other Use Cases

The motivating scenario is a demonstration of what LongTale can do to solve a tricky problem that other tools cannot. LongTale can also do more. In this section, we describe two additional use cases to show what else can LongTale do and its potential to push the limits of the state-of-the-art automatic performance-diagnosing tools.

6.5.1 Different Performance Interference Factors. As a highly distributed system, a microservices application may suffer from multiple, different performance issues at different levels of severity—but how much is a single source of interference contributing to the slow down? In other words, can we quantitatively show how likely it is that a particular performance interference factor is slowing down the request, and how severe it is? To answer this question, we extend the aforementioned regression method to cover factors that are beyond the single factor that contributes most to the abnormal long tail. We perform regression throughout the whole population of the requests, instead of picking the head and the tail. We divide the whole CDF into 20 buckets of latency segments: $\leq$ 5%, 5–10%, 10–15%, etc. In terms of implementation, the only parameter changed from the previous scenario is in the `policy materializer`: we generate spanID sets for each of the 20 buckets, instead of two.

In this experiment, we inject 5 different kinds of anomalies, one anomaly on each of five VMs. Table 5 shows how we deploy the anomalies. We perform two experiments with different anomaly severities, using "kernel decompression" (No. 1) as a reference (i.e., not changing it). We intentionally tune down the intensity of No. 2 and No. 3 while tuning up No. 4 and No. 5. Figure 8 shows the importance of each interference factor in the two experiments. It shows the top six features of each explanation result, with their coefficient values normalized to between the greatest coefficient and the one immediately less than the 6th-biggest coefficient. The coefficient change for each anomaly between the two experiments matches our operational behavior of tuning up or down the anomalies' severity. We can see in experiment *a* that kernel compilation is considered to be the factor with the highest impact as it executes all the time. But in experiment *b*, the kernel decompression (`gzip`) takes the top place. It also shows that, with the same pace of web traffic load, `apache2` impacts service latency more severely than `nginx` in both cases, which is in line with our perception of the lightweight nature of `nginx` server.

6.5.2 Assistance for In-depth Analysis. Sometimes, a diagnostician may want to discover which subsystem in an application has the

Table 5: Different performance interference factors: D_a and D_b indicate the duration of the anomaly; I_a and I_b are the interval of each microburst.

ID	Process	Deployment	D_a	I_a	D_b	I_b
1	tar/gzip	kernel decompression	12s	20s	12s	20s
2	stress-ng	CPU saturation	5s	20s	5s	40s
3	cc1	kernel compilation	all	0s	10s	10s
4	apache2	ab w/ 1k users	5s	20s	5s	10s
5	nginx	ab w/ 1k users	5s	20s	5s	10s

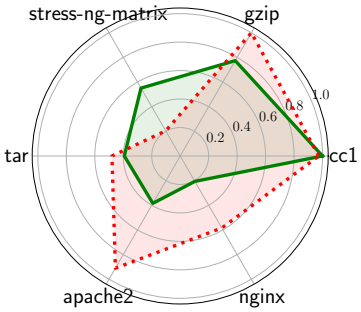


Figure 8: Severity of different kinds of performance interference factors: the green solid line shows experiment *a*, and the red dotted line shows experiment *b*.

most influence on a slowdown. In this experiment, we inject an anomaly that happens on the execution path itself. We simulate a real-world scenario that happened in the Google Gmail back end [11], in which bursty function calls on the path of an RPC saturate the CPU, leading to performance regression. In our experiment, we inject a compute-intense function, `stressMatrix` (multiplying two 100×100 matrices), that is occasionally called on the path of the `login` span in the `user` service. The burst happens every 100 seconds with duration of 5 seconds. To explain which function in a process contributes the most to the slowdown, LongTale narrows the scope to expand stacks only for the process of interest. We pick 20 as the stack depth for this experiment, which covers most of the function calls. After going through the data-processing pipeline of LongTale and using the vectors of the expanded stacks as features, LongTale correctly highlights `stressMatrix` as the greatest contributor to the slowdown.

To better help the analyst understand how the anomalous function was called, LongTale can generate a flame graph [21] to visualize the aggregated call stacks. A flame graph represents a collection of stack traces. The bars at the "top" of the flame show the functions that were executing when the various stack traces were captured. The underlying bars show the functions in the captured call stacks. The width of bar reflects the fraction of the time that a given call stack (the bar's function, and the calling functions under that bar) appears in the sampled stack traces.

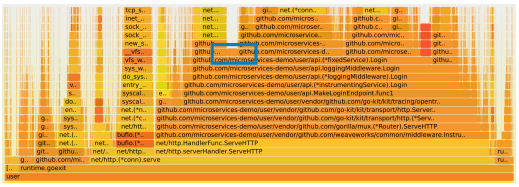


Figure 9: Re-aggregated flame graph for fast spans.

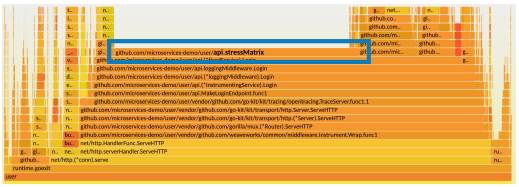


Figure 10: Re-aggregated flame graph for slow spans.

Figures 9 and 10 are flame graphs generated for the 98%-percentile-up and the 20%-percentile-down spans using the flame graph generation tool [21]. The blue box in the middle of Figure 10 highlights `stressMatrix`. It takes 42.16% of the CPU. The blue box in the middle of Figure 9 also highlights the `stressMatrix` function (the bar is too short to show the function name), and it takes only 1.58% of the CPU. This difference in key contributing features between the re-aggregated slow and fast stack traces is the reason why LongTale can pinpoint the culprit by running linear regression over randomly generated vectors. Through the flame graph, an analyst can see how `stressMatrix` was called. A de facto practice in industry is to generate a flame graph when a new version of software is deployed [25], so that when a performance regression occurs, analysts can revisit the flame graphs of previous versions. The generation of flame graphs in LongTale both (1) supports this best practice and (2) pushes the state of the art by recording both fast and slow spans in a highly distributed setting, instead of just for a single host.

7 RELATED WORK

There are various performance analysis tools and tracing systems attempting to mitigate performance issues. We categorize them in three groups and clarify how LongTale differentiates itself from these state-of-the-art proposals.

End-to-end application-layer tracing. This is the most mature performance debugging technology for distributed applications. X-Trace [18] and Dapper [35] are the earliest proposals pioneering the concept of Traces and Spans for distributed tracing. Pivot Tracing [20] extends these early works with the "happened-before join" operator to incorporate causal tracing in end-to-end analysis. Canopy [26] enhances traditional application-layer tracing by decoupling tracing components and a pipeline of tracing data transformation. All these works on distributed tracing focus on userspace activities that happen on the execution path of applications, but not analyze the whole system state across applications and the underlying operating system. This limits their applicability for off-path event analysis.

Table 6: Comparison of root cause analysis approaches for microservices.

System	Localization granularity	Detectable anomaly duration	No tracing instrumentation	Execution state analysis	Fine-grained root cause explanation
MicroRCA [41]	service-level	seconds	✓	✗	✗
MicroCause [29]	service-level	seconds	✗	✗	✗
ε-Diagnosis [33]	service-level	seconds	✓	✗	✗
Wang et al.[37]	service-level	seconds	✓	✗	✗
LongTale	function-level	milliseconds	✓	✓	✓

Cross-layer tracing. Lacking visibility into the kernel and other external factors impedes distributed system analysis from solving trickier performance interference problems induced outside of the services under examination. To improve system-level visibility in performance troubleshooting, there have been efforts to incorporate kernel behaviors in the analysis. Ardelean et al. [11] propose "vertical context injection" by passing application context information to the kernel via the `getpid`'s system call arguments. Sheth et al. [34] rely on a custom kernel module to incorporate system call information as a new layer of tracing within application-level tracing. Zeno [43] collects network packets leaving and arriving at a node to facilitate temporal causality analysis for off-path events. Seer [19] relies on multiple kernel metrics combined with application-layer tracing to relieve resource saturation problems in microservices. All these works require modifications on either the operating system kernel or an application-layer tracing framework to collect specific types of traces to enable cross-layer tracing, which has both security and performance implication. LongTale, on the other hand, takes a purely data-oriented approach, where cross-layer traces are collected and stitched for informative troubleshooting *without* any modification on the kernel subsystems or an application-layer tracing framework. This very data-oriented nature enables LongTale to be easily extended to stitch different types of performance metrics beyond stack traces.

Host-based performance profiling. There are numerous system-level performance profiling and tracing tools [2, 6–9, 17] used for performance troubleshooting. Coupled with effective visualization aids such as flame graphs [21], they enable an analyst to interpret the lower-level execution state of a given host or a particular process, and conduct comparative analysis for different (normal/abnormal) periods. The limitation of such approach, however, is that it only represents a host-centric view, and at a coarse-grained time granularity, Real-world microservice deployments may experience abnormal long-tail latency caused by sparsely scattered microbursts in resource usage. These types of anomalies cannot easily be analyzed by conventional methods such as flame graph-like data visualization as the microburst events will not stand out in the visualization. LongTale builds on the fine-grained observability of the existing host-centric profiling tools and advances interpretation of the collected data by combining them with end-to-end application-layer tracing and machine learning driven analysis.

Root cause localization in microservices. There are many works on root cause analysis for failures and security incidents in distributed systems [27, 42, 44, 45]. Few of them are dedicated to pinpointing the root cause of performance problems in microservices. The most recent and closely related works are MicroRCA [41], MicroCause [29], ε-Diagnosis [33], and Wang et al. [37]. In these works, the authors identify causality relationships between key performance indicators and relevant system resource metrics (e.g., CPU/memory utilization). Ultimately, the resource metrics that are highly correlated with abnormal microservice performance are presented as a possible explanation for the anomaly. There are two limitations in such approaches. First, due to the coarse-grained measurement period of resource metrics (e.g., seconds), their causality analysis cannot diagnose the kind of short-lived microbursts captured by LongTale. The usefulness of their anomaly explanation is also limited; without any execution state analysis, resource metrics alone do not provide any further detail on actual culprits. LongTale, on the other hand, can point an analyst directly to a culprit (e.g., responsible process or kernel component), as well as provide an intuitive visual aid for the analyst to navigate the temporal and spatial distribution of the anomaly, making its diagnosis results immediately useful. Table 6 summarizes the feature comparison among them. Besides the advanced research works in academia, there are also many Application Performance Monitoring (APM) products that can perform root cause analysis to an extent, e.g., Dynatrace [15], Datadog [14], Elastic [16], and AppDynamics [10]. They are mainly built upon graph analysis over the dependency of services at container level to provide diagnostic insights, which is coarse grained. So far, none of them can perform automatic explanation for millisecond-level sporadic microbursts of performance anomalies.

8 DISCUSSION

Not all performance anomalies reflect their behaviors in stack traces that LongTale relies on. Below, we discuss what types of performance anomalies LongTale can and cannot diagnose.

Applicable scenarios. At least four types of abnormal behaviors can be reflected in the stack traces: (i) CPU intensive incidents, either in user space or in kernel space, that interfere with the performance of critical tasks; (ii) anomalies induced by high I/O activities, whose behaviors can be characterized by the user and kernel threads handling disk and network I/O interrupts; (iii) memory-intensive anomalies, where the kernel swapping routine frequently kicks in for memory paging; and (iv) hardware failures that force the kernel or user-space threads to retry failed operations until success. All these scenarios activate the CPU to perform additional tasks as a

response, which will be captured by LongTale as dominant features during root cause explanation.

Inapplicable scenarios. There are many performance incidents rooted in external component failures. The behavior of these external components is not captured by the stack traces, and thus not analyzable by LongTale. For example, external networking issues (e.g., network hardware failure or misconfiguration) will not show any traces in cross-layer stitching. As another example, a crashed DNS server failing to resolve domain name will introduce abnormal long latency. Yet, because DNS queries are relatively rare, this kind of repeated queries may not be captured and will not stand out in stack traces. Also, the type of anomalies that manifest themselves slowly over time cannot be diagnosed by LongTale. For example, a long-lasting daemon process may suffer from a memory leak without being detected because memory leaks slowly over time, and the daemon may not be actively occupying the CPU. In that case, their function symbols will unlikely be captured in sampled stack traces, and thus will not be highlighted by LongTale.

9 CONCLUSION

As the microservices-based architecture enables high scalability, modularity, and flexibility for cloud-based applications, performance diagnosing tools should evolve to support better observability and debuggability. LongTale combines two widely adopted application- and system-level tracing to enable automatic performance explanation. Experiments show that LongTale can pinpoint different anomalies in an application-agnostic, immediately deployable fashion.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments and help in improving this paper. This material is based upon work supported in part by the National Science Foundation under Grant Numbers 1642158 and 1743363.

REFERENCES

[1] 2015. Adopting Microservices at Netflix: Lessons for Team and Process Design. https://www.nginx.com/blog/adopting-microservices-at-netflix-lessons-for-team-and-process-design/.
[2] 2017. ktap. https://github.com/ktap/ktap.
[3] 2019. Jaeger. https://jaegertracing.io.
[4] 2019. OpenZipkin. https://zipkin.io.
[5] 2019. sklearn.linear_model.ElasticNet. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.ElasticNet.html.
[6] 2020. bpftrace. https://github.com/iovisor/bpftrace.
[7] 2020. Intel VTune Profiler. https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html.
[8] 2020. LTTng. https://lttng.org.
[9] 2020. SystemTap Wiki. https://sourceware.org/systemtap/wiki.
[10] AppDynamics. 2022. AppDynamics: The world's #1 APM solution. https://www.appdynamics.com/.
[11] Dan Ardelean, Amer Diwan, and Chandra Erdman. 2018. Performance Analysis of Cloud Applications. In *Proc. USENIX NSDI.*
[12] Wensley Bart. 2019. Mark pods as not ready when host goes offline. https://opendev.org/starlingx/nfv/commit/cdd6c334d9d1d6c0f4de344fff8ef2af28c76e56.
[13] Melanie Cebula. 2017. Airbnb, From Monolith to Microservices: How to Scale Your Architecture. FutureStack17.
[14] Datadog. 2022. Datadog: Cloud Monitoring as a Service. https://www.datadoghq.com/.
[15] Dynatrace. 2022. Dynatrace: The Leader in Automatic and Intelligent Observability. https://www.dynatrace.com/.
[16] Elasticsearch B.V. . 2022. Elastic APM. https://www.elastic.co/guide/en/apm/index.html.
[17] Stephane Eranian. 2019. Linux kernel profiling with perf. https://perf.wiki.kernel.org/index.php/Tutorial.
[18] Rodrigo Fonseca, George Porter, Randy H Katz, Scott Shenker, and Ion Stoica. 2007. X-trace: A Pervasive Network Tracing Framework. In *Proc. USENIX NSDI.*
[19] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *Proc. ACM ASPLOS.*
[20] Mohamad Gebai and Michel R. Dagenais. 2018. Survey and Analysis of Kernel and Userspace Tracers on Linux: Design, Implementation, and Overhead. *Comput. Surveys* 51, 2 (2018).
[21] Brendan Gregg. 2016. The Flame Graph. *Commun. ACM* 59, 6 (2016).
[22] Brendan Gregg. 2018. Linux Extended BPF (eBPF) Tracing Tools. http://www.brendangregg.com/ebpf.html.
[23] Brendan Gregg. 2019. *BPF Performance Tools.* Addison-Wesley Professional.
[24] Brendan Gregg. 2019. perf Examples. http://www.brendangregg.com/perf.html.
[25] Brendan Gregg. 2019. USENIX ATC 2017 Invited Talk: Visualizing Performance with Flame Graphs. https://youtu.be/D53T1Ejig1Q.
[26] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. 2017. Canopy: An End-to-End Performance Tracing and Analysis System. In *Proc. ACM SOSP.*
[27] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. 2018. Towards a Timely Causality Analysis for Enterprise Security.. In *Proc. NDSS Symposium.*
[28] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2018. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. *ACM Transactions on Computer Systems (TOCS)* 35, 4 (2018), 11.
[29] Yuan Meng, Shenglin Zhang, Yongqian Sun, Ruru Zhang, Zhilong Hu, Yiyin Zhang, Chenyang Jia, Zhaogang Wang, and Dan Pei. 2020. Localizing Failure Root Causes in a Microservice through Causality Inference. In *Proc. IEEE/ACM International Symposium on Quality of Service.*
[30] Sam Newman. 2015. *Building Microservices: Designing Fine-Grained Systems.* O'Reilly Media, Inc.
[31] Matt Ranney. 2016. What I Wish I Had Known before Scaling Uber to 1,000. In *Proc. GOTO Conference Chicago.*
[32] Robert Ricci, Eric Eide, and the CloudLab Team. 2014. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *; login:* 39, 6 (2014), 36–38.
[33] Huasong Shan, Yuan Chen, Haifeng Liu, Yunpeng Zhang, Xiao Xiao, Xiaofeng He, Min Li, and Wei Ding. 2019. ε-Ddiagnosis: Unsupervised and real-time diagnosis of small-window long-tail latency in large-scale microservice platforms. In *The World Wide Web Conference.* 3215–3222.
[34] Harshal Sheth and Andrew Sun. 2018. Skua: Extending Distributed Tracing Vertically into the Linux Kernel. In *Proc. DevConf.*
[35] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.* Technical Report. Google, Inc. https://ai.google/research/pubs/pub36356
[36] The OpenTelemetry Authors. 2022. OpenTelemetry: High-quality, ubiquitous, and portable telemetry to enable effective observability. https://opentelemetry.io/.
[37] Lingzhi Wang, Nengwen Zhao, Junjie Chen, Pinnong Li, Wenchi Zhang, and Kaixin Sui. 2020. Root-cause metric location for microservice systems via log anomaly detection. In *2020 IEEE International Conference on Web Services (ICWS).* IEEE, 142–150.
[38] Qingyang Wang, Yasuhiko Kanemasa, Jack Li, Deepal Jayasinghe, Toshihiro Shimizu, Masazumi Matsubara, Motoyuki Kawaba, and Calton Pu. 2013. Detecting transient bottlenecks in N-tier applications through fine-grained analysis. In *Proc. IEEE ICDCS.*
[39] Weaveworks, Inc. 2018. Sock Shop – A Microservice Demo Application. https://microservices-demo.github.io.
[40] Daniel Weibel. 2019. How to autoscale apps on Kubernetes with custom metrics. https://learnk8s.io/autoscaling-apps-kubernetes.
[41] Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. 2020. MicroRCA: Root Cause Localization of Performance Issues in Microservices. In *Proc. IEEE/IFIP NOMS.*
[42] Yang Wu, Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2017. Automated Bug Removal for Software-Defined Networks. In *Proc. USENIX NSDI.*
[43] Yang Wu, Ang Chen, and Linh Thi Xuan Phan. 2019. Zeno: Diagnosing Performance Problems with Temporal Provenance. In *Proc. USENIX NSDI.*
[44] Wenchao Zhou, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. 2011. Secure Network Provenance. In *Proc. ACM SOSP.*
[45] Wenchao Zhou, Suyog Mapara, Yiqing Ren, Yang Li, Andreas Haeberlen, Zachary Ives, Boon Thau Loo, and Micah Sherr. 2012. Distributed Time-aware Provenance. *Proceedings of the VLDB Endowment* 6, 2 (2012), 49–60.
[46] Hui Zou and Trevor Hastie. 2005. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: series B (Statistical Methodology)* 67, 2 (2005).