Aggregate-Driven Trace Visualizations for Performance Debugging

Vaastav Anand[1], Matheus Stolet[1], Thomas Davidson[2], Ivan Beschastnikh[1], Tamara Munzner[1], and Jonathan Mace[2]

[1]The University of British Columbia
[2]Max Planck Institute for Software-Systems

Abstract

Performance issues in cloud systems are hard to debug. Distributed tracing is a widely adopted approach that gives engineers visibility into cloud systems. Existing trace analysis approaches focus on debugging single request correctness issues but not debugging single request performance issues. Diagnosing a performance issue in a given request requires comparing the performance of the offending request with the aggregate performance of typical requests. Effective and efficient debugging of such issues faces three challenges: (i) identifying the correct aggregate data for diagnosis; (ii) visualizing the aggregated data; and (iii) efficiently collecting, storing, and processing trace data.

We present TraVista, a tool designed for debugging performance issues in a single trace that addresses these challenges. TraVista extends the popular single trace Gantt chart visualization with three types of aggregate data - metric, temporal, and structure data, to contextualize the performance of the offending trace across all traces.

1 Introduction

Prevalent cloud system designs, like microservices and serverless, increase the complexity of systems by spreading the execution of a request across many machines. This impedes the operators from diagnosing performance problems, as it requires deducing root causes from distributed symptoms [5, 19]. Distributed tracing is as an effective way to gain visibility across distributed systems [10, 18, 19]. Distributed tracing frameworks, like OpenTelemetry [7], Jaeger [12], and Zipkin [27], are used by most major internet companies [6, 13, 24].

Distributed tracing tools arose out of a need to understand the behavior of *individual requests*: identifying the services invoked, diagnosing problematic requests, and debugging correctness issues [9, 10, 24]. Thus, the user interfaces and visualizations in distributed tracing tools center on providing an in-depth view into the execution of a single request. Diagnosing *system-wide* cloud performance issues remains a human-driven task [5] as requests must be aggregated and compared to explain performance anomalies [8]. This is because a request has a performance anomaly only with respect to *other* requests.

Aggregate analysis of tracing data has been successfully used for accomplishing specific tasks where the relevant aggregate analysis task is known *a priori*. For example, modeling workloads and resource usage [3, 4, 20, 26], and outlier detection [2, 15, 28]. However, diagnosis of performance issues requires general purpose aggregation of trace data as a user must evaluate different hypotheses for the root cause of the issue, with each hypothesis potentially requiring a different form of trace aggregation.

Such general purpose aggregation for effective performance debugging is non-trivial for three reasons: (i) the aggregate data required is not known a priori; (ii) designing a visualization tool that effectively supports exploration of complex traces is non-trivial; and (iii) designing an efficient backend for collecting, storing, and processing trace data is hard. Existing trace analysis systems, like Canopy [13] have focused primarily on the third challenge and now can efficiently process billions of traces per day. However, analysis of this data for performance debugging requires a user to sift through this large amount of data. This requires building efficient visualization tools that can support general purpose aggregate queries. This problem is conceptually and computationally intractable without having any a priori knowledge about the kind of queries a user might perform. To this effect, existing visualization tools take a simple approach by only showing graphs of high level metrics. However, this approach is not sufficient for performance debugging as that requires jointly considering aggregate data across the temporal, structural, and metric dimension of the traces.

To addresses these challenges, we present our initial work on *TraVista*: a tool for distributed traces designed for diagnosing performance issues. TraVista builds on the prevalent approach of visualizing a single request by augmenting this view with information to aid the user in contextualizing the performance of one request with respect to other requests. TraVista constrains the possible aggregations to those that are relevant to spans, events, and relation-

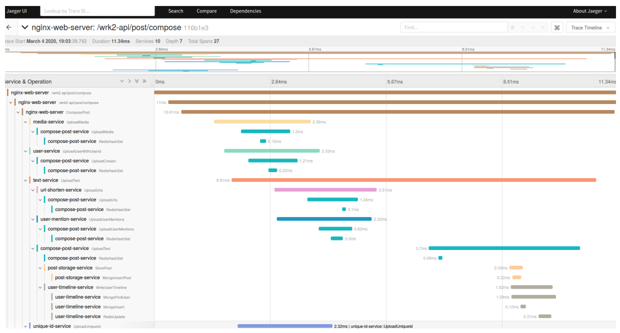
Jaeger UI
Lookup by Trace ID...
Search
Compare
Dependencies
About Jaeger
nginx-web-server: /wrk2-api/post/compose 116b1e3
Find...
Trace Timeline
race Start March 4 2020, 19:03:39.742 Duration 11.34ms Services 10 Depth 7 Total Spans 27
ervice & Operation
0ms
2.84ms
5.67ms
8.51ms
11.34ms
nginx-web-server /wrk2-api/post/compose
nginx-web-server /wrk2-api/post/compose
11ms
nginx-web-server ComposePost
10.61ms
media-service UploadMedia
2.36ms
compose-post-service UploadMedia
1.2ms
compose-post-service RedisHashSet
0.15ms
user-service UploadUserWithUserId
2.33ms
compose-post-service UploadCreator
1.21ms
compose-post-service RedisHashSet
0.22ms
text-service UploadText
8.91ms
url-shorten-service UploadUrls
2.51ms
compose-post-service UploadUrls
1.24ms
compose-post-service RedisHashSet
0.1ms
user-mention-service UploadUserMentions
2.32ms
compose-post-service UploadUserMentions
0.82ms
compose-post-service RedisHashSet
0.3ms
compose-post-service UploadText
3.7ms
compose-post-service RedisHashSet
0.09ms
post-storage-service StorePost
0.33ms
post-storage-service MongoInsertPost
0.22ms
user-timeline-service WriteUserTimeline
1.02ms
user-timeline-service MongoFindUser
1.09ms
user-timeline-service MongoInsert
0.12ms
user-timeline-service RedisUpdate
0.31ms
unique-id-service UploadUniqueId
2.32ms | unique-id-service::UploadUniqueId

Figure 1: Gantt chart visualization from Jaeger [12].

ships present in the trace, instead of considering all possible aggregations across all traces. Thus, TraVista builds on the established Gantt chart visualization of a single trace as it provides a familiar starting point to distributed tracing users and narrows the scope of aggregation information to three kinds: metric, temporal, and structural aggregation.

2 Background & Motivation

We briefly review the background on distributed tracing and describe three challenges to building trace analysis tools for performance debugging.

Trace Structure A trace is a Directed Acyclic Graph (DAG) of *spans*. A span can be thought of as a particular task that a system performs to execute a request. The granularity of the task is user-defined and controlled. A span can represent many things, including a single function execution, a single thread execution, or a single operating system process comprised of multiple threads. Within a span, developers can also add *events*, which are typically unstructured log messages. Span annotations and events are developer-defined and are connected together with *edges*, representing happened-before relationships [16]. Edges *between* spans typically represent network communication (e.g., the send and receive of RPCs). Each span records its timing and duration, as well as arbitrary key-value annotations provided by a developer: such as logging a span's arguments. Each individual trace can be very large, comprising thousands of spans and events [8, 13, 17], and production systems capture traces for millions of requests per day [13].

Most distributed tracing frameworks represent traces using spans [7, 12, 24], but some frameworks are based only on events [10]. For event-based frameworks, it is straightforward to group events together into spans (*e.g.* events in the same thread). We use the term *task* to refer to both of these concepts. In span-based tracing frameworks a task corresponds to a span and in event-based tracing frameworks a task corresponds to a collection of events.

Gantt chart Visualization The Gantt chart is the most popular way to represent a trace. A Gantt chart provides a concise view of all the tasks in a trace, and their respective latencies. Figure 1 shows a Gantt chart screenshot from Jaeger [12]. Jaeger provides a minimal trace visualization: time is represented on the x-axis, each task is represented as a single lane, and the length of the lane encodes the task's duration. X-Trace [10] extends this by superimposing events on task lanes using the event's timestamp as its x-axis position. The visualization also connects events with edges to show the recorded happened-before relationships [16]. Both visualizations help users understand how a request executes in the system, and for debugging errors in a request.

2.1 Aggregate Analysis Challenges

We define *aggregate analysis* as analysis of data collected and aggregated from a set of traces to answer questions about the state, health, and properties of the distributed system being traced. These questions can be high-level: "Is the system running correctly?", or narrowly focused: "Why did a trace have such high latency?". We identify three key challenges of building aggregate analysis tools for tracing data.

Choosing Aggregate Data Traces are richly annotated graph-structured data and it is not obvious which dimensions or subset of the data is relevant for finding the root cause of performance anomalies. Pintrace [14], the distributed tracing system at Pinterest, uses aggregate analysis to compare two different groups of traces to narrow down the root cause of an error. However, the granularity here is coarse as the comparison is only performed on the distribution of latencies of the traces in each group. This is only useful for identifying high-level trends. Early work on trace comparison similarly compared latency distributions, but required traces to be structurally isomorphic [22, 23]; this is rare in practice as most traces are structurally unique [17].

Visualizing the data Building an effective visualization tool is difficult, especially for high-dimensional data such as trace data. It is not obvious how to present the data to developers for effective analysis or debugging. The Gantt chart visualization is prevalent for visualizing a single request [12, 25]. However, the main drawback of a Gantt chart is its focus on individual requests [25]. It provides no context for whether the trace represents normal or outlier behavior – a feature intrinsically dependent on *other* traces. This lack of context makes Gantt charts difficult to use for aggregate analyses. Recent work at Uber [8] compares the structure of incoming traces with a "good" set of traces to find "bad" traces, and then visualizes their difference as a directed graph. Canopy [13] can generate simple graphs of derived metrics from traces. However, none of the existing tools jointly visualize aggregate structural, metric, and temporal data. Designing effective visualizations for aggregate analysis continues to elude the tracing community [25].

Data processing system A key challenge for developing aggregate visualizations is designing an efficient backend system to support filtering and aggregating trace data across multiple dimensions. In particular, it is computationally expensive to filter or group traces based on properties of their graph structure (*e.g.* filtering traces that contain a specific sequence of tasks). Canopy [13] feature extraction is a post-processing step before storing traces. Canopy supports database queries over these features, but its statistical analysis workflow does not incorporate trace structure. The authors acknowledge that inspecting the structure of traces is a separate workflow handled by a different tool and visualization.

3 TraVista Goals and Design

Our goal with TraVista is to help developers working on distributed systems to diagnose performance issues in traces. A key intuition behind TraVista is to extend existing single-trace visualizations; in particular, the Gantt chart visualization. Doing this helps address all three challenges listed in Section 2.1. First, a Gantt chart is a well-established visualization that is a familiar starting point for existing distributed tracing users. Second, this narrows the scope of data to be presented, to only the data that is relevant to the selected trace. This makes it easier for us to aggregate relevant features in the dataset to compare the selected trace to other traces in the system, and easier to design an efficient backend, since it only needs to support specific kinds of queries.

3.1 Aggregation Types

In the context of visualizing a single trace, there are three kinds of aggregated data that we consider along three different dimensions - tasks, events, and edges. We chose these dimensions because they provide enough information for us to understand the structural and temporal layout of the traces in the dataset, and the underlying metrics, whilst allowing an efficient visualization and backend to be built.

Metrics Different systems have different **performance metrics** to quantify the health of their services. Common examples include request latency, memory utilization, and throughput, though many systems also define application-level metrics. Aggregating a metric can mean calculating a statistical quantity or storing the entire distribution. Performance metrics can also be broken down using various filters such as a service name, host name, geographical location, etc. TraVista does metric aggregation for tasks and presents the metric distribution across all traces for each task in the trace.

Temporal Aggregation At any given point in time there may be multiple requests being serviced by the system. The information present in a trace is limited to a single request and does not contain information about concurrently executing requests. The root cause of outliers can sometimes be concurrent work, *e.g.* high latency can be caused by resource contention. Zeno [29] uses a similar idea for performance debugging but solely focuses on identifying queueing delay. TraVista generalizes this approach by using temporal data to identify other types of resource contention. Currently, TraVista uses the number of requests being handled by a process executing a task to approximate the contention during the execution of that task.

Structural Aggregation Traces are rich sources of structural information about how different services communicate with each other. Structural differences between traces can be very useful in helping users identify the root cause of an anomaly, *e.g.* occurrence of unlikely sequences of events or edges in a trace can highlight and explain abnormalities in a trace. TraVista uses structural aggregation for both edges and events. To show anomalous events in tasks, it currently shows the frequency of each event occurring in tasks of that type, and emphasizes events that occur with low frequency. TraVista uses structural information to display the frequency that a task invokes its child tasks, and emphasizes edges that occur with low frequency.

4 TraVista Interface

TraVista comprises of a frontend NodeJS application for visualizing traces and a backend server for storing and querying trace data. To guide users towards potential root causes, TraVista makes use of three visualization principles: *eyes beat memory*, the *popout phenomenon*, and *detail on demand* [21]. We describe these principles inline with our TraVista description below. For clarity, we describe TraVista's visualization within the context of diagnosing a high latency trace from an open source X-trace dataset [1] consisting of traces from the DeathStar social network microservice benchmark [11].

Example Figure 2 shows an annotated screenshot of TraVista's trace visualization. The trace displayed in Figure 2 is a high-latency "ComposePost" request. The root cause of this request's high latency is a delay in the Redis cache that stores user timelines. We will refer to the circled numbers in the figure (*e.g.*①) throughout this section.

Gantt Chart The starting point of TraVista is the Gantt chart visualization ①, similar to the visualization shown in Figure 1. The x-axis represents time, each lane represents a task, and the length of the lane represents the duration of the task. The Gantt chart gives users an immediate overview of the structure of a trace and the duration of the different tasks within the trace relative to each other. TraVista uses the Gantt chart as its starting point because it is a familiar representation for trace data.

Metric Histograms We augment the Gantt chart with a list of histograms ② showing the metric distributions of the different tasks within the trace. The metric shown is latency as we are trying to diagnose the source of the high latency of the trace. Each histogram corresponds to one task type (*e.g.* a specific service) and plots its latency distribution across all traces. Latencies are binned on the x-axis, and y-axis frequencies are plotted in log-scale, since

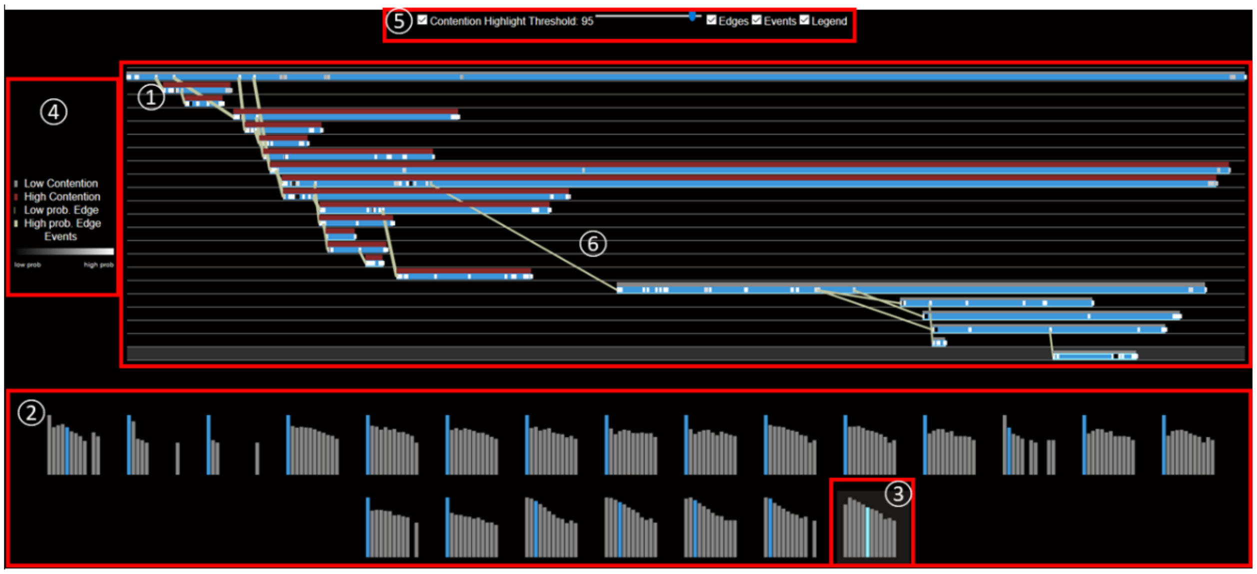

Figure 2: TraVista's Gantt chart visualization extended with aggregate data.

smaller values are intuitively more interesting and must be visually identifiable. As the y-axis is log-scale, any value away from the peak of the distribution is significantly away from the median of the distribution and is thus potentially anomalous. The blue highlighted bar in each histogram indicates where the current trace's task falls in the distribution. Histograms are ordered left-to-right in the same order as the tasks are displayed top-to-bottom in the Gantt chart. When a user hovers the mouse over a specific histogram ③, TraVista highlights the corresponding task in the Gantt chart. In the example, ③ is the "WriteUserTimeline" task, which has substantially higher latency than normal.

TraVista displays latency histograms for all tasks simultaneously, adopting the principle of *eyes beat memory* [21, Ch. 6.5]. Using eyes to switch between views that are simultaneously visible has lower cognitive load on the user than consulting memory. Reducing cognitive load is important because human working memory is a limited resource, and when its limits are reached, the user fails to absorb the presented information.

Outlier Events Within the Gantt chart ①, TraVista augments each lane with two additional sources of information. The first is information about events that occurred during the task. Figure 3 shows an enlarged view of a lane with this additional information. TraVista represents events as vertical lines overlaid ⑦ on the blue rectangles used to represent each task. The x-position of an event encodes the event's timestamp.

TraVista utilizes the *popout phenomenon* [21, Ch. 5.5.4] to aid users in identifying uncommon events by encoding rare events (*i.e.*, events that do not frequently occur in a task) with low luminance (black). Events that occur frequently are encoded with high luminance (white). This enables users to quickly identify tasks that have many outlier events. Users can then dive deeper into the specific rare events. In the highlighted task in Figure 2, two events have low luminance. Both of these correspond to redis updates.

Resource Contention The second type of information that TraVista augments the lane in the Gantt chart with is *resource contention*. We design a novel visualization called *molehills* ⑧ that allows the user to visualize potential resource contention occurring as a result of other tasks executing on the same process at the same time. This data is gathered at a millisecond level granularity and linearly scaled based on the maximum value across the trace, where it is plotted as a bar chart on top of every span. TraVista highlights periods where contention is higher than a user-defined percentage threshold ⑤ in red.

Outlier Edges TraVista augments the Gantt Chart with edges ⑥ between spans which represent happened-before relationships [16]. It utilizes the popout phenomenon [21, Ch. 5.5.4] , this time utilizing the size channel rather than the color channel, to aid users in identifying uncommon edges by encoding uncommon edges with smaller width and common edges with bigger width. This allows the users to quickly identify thin edges as the relationships that are structurally uncommon across the traces.

Detail Since traces can be very detailed, visualizing a trace can lead to significant complexity. Augmenting a trace with aggregate information increases this complexity. By default, contention, events, edges, and the legend ④ are disabled, and left as selectable checkboxes allowing users to request *detail on demand* [21, Ch. 6.7].

5 TraVista Backend

Our approach for building the backend for TraVista is to follow a simple backend design that allows for efficient querying of aggregated data. Like Canopy [13], when a

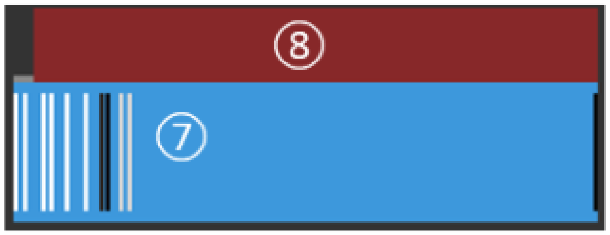

Figure 3: Zoomed-in view of a lane in TraVista's Gantt chart.

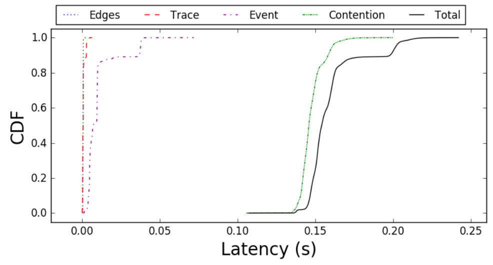

Figure 4: Latency breakdown for loading aggregate data.

new trace is received by the TraVista backend, we extract features and metrics necessary for the visualization. These features are stored in database tables with appropriate indexes for querying. This approach is sufficient for modest datasets like the DeathStarBench [11] X-Trace dataset [1]. The dataset consists of 22,286 traces (totaling 147,812 tasks and 2,215,250 events). Pre-processing a trace to extract the relevant features takes around 1ms. Figure 4 plots the CDF of the backend latency breakdown for loading different kinds of data - raw trace, event aggregate data, edge aggregate data, and contention aggregate data for each trace in the dataset. Backend latency for a trace is primarily dependent on the latency of loading contention aggregate data. However, TraVista's current backend approach will not scale over time. Figure 5 shows how the latency for loading the aggregate data increases linearly with the total number of traces in the database. To perform this experiment, we generated synthetic copies of the X-Trace dataset and measured the latency breakdown for the biggest trace in the dataset 1000 times.

Despite the scalability concern with our simple backend, we believe that bounding the types of aggregations to those relevant to the data in a single trace can greatly simplify the design of an efficient backend. We believe that a backend designed to support the performance debugging visualizations will yield the best results in efficiency and efficacy for performance debugging in the future.

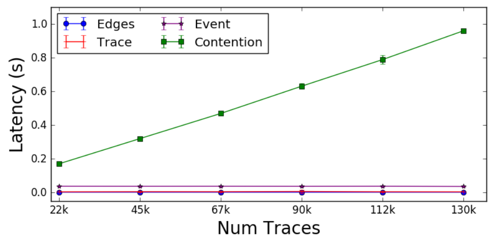

Figure 5: Average Latency breakdown for loading a single trace with different number of traces in the database.

6 Challenges & Future Work

User Study Evaluation With TraVista, broad user adoption is one of our core goals, and we believe that achieving this requires treating usability and visualization as first-class concerns. We will collect data about user experiences of using TraVista with real production systems by running a user-study with production users of Jaeger.

Trace Size The major drawback of the Gantt Chart visualization is that it does not scale well with the number of tasks in a trace as it is unable to fix all the tasks on the screen. "Scrolling" is a commonly used tactic to counter this problem but it is a bad solution as it prevents the user from getting the full view of a trace. Moreover, scrolling does not fit with TraVista's visualization idioms. To fix this, we will use *detail on demand* to first show a compact picture of the trace on the screen followed by using *semantic zooming* [21, Ch. 11.5.2] to allow the user to zoom into different sections of the trace enabled with TraVista's visualizations. To help the user decide as to which part to zoom into, we will show the latency histograms for the top n tasks whose latency deviates from the median.

Aggregate Analysis of Traces TraVista is use-case driven, and we followed a 'path of least resistance' by augmenting the Gantt chart visualization. We do not believe TraVista will solve all aggregate analysis use cases, and we are interested in exploring other aggregate analysis use cases where this approach will not work. For example, TraVista would not be directly useful for tasks like clustering or trace comparison.

Backends for Complex Data Since traces are large and complex graphs, it is infeasible to support real-time queries of arbitrary structural properties. However, it remains unclear to us whether such queries are actually *necessary* for problem diagnosis. In TraVista, our goal is to develop a specialized backend, that only supports the kinds of queries needed by the frontend. Since we know ahead of time the kinds of aggregations we need to do for tasks and events, the backend can maintain and update those aggregations with each incoming trace. Eventually, when we need to support more complex filtering and grouping, this approach may also not scale. The solution may require a trade-off between pre-calculated aggregates and on-demand queries. Our intuition to address this is that our tools do not require perfect results, and can tolerate a high margin of error. Our goal is to support human driven diagnosis, and the ultimate measure of success is not accurate statistics, but facilitating quicker problem diagnosis. Techniques such as sub-sampling will be helpful for reducing backend costs.

We believe that TraVista represents a step towards better tools and visualizations for aggregate trace analysis. TraVista is ongoing work, and in future we will incorporate arbitrary metrics, filtering, and our zooming solution for traces with large number of microservices.

References

[1] V. Anand and J. Mace. X-Trace DeathStarBench Dataset. Retrieved October 2019 from `https://gitlab.mpi-sws.org/cld/trace-datasets/deathstarbench_traces`.

[2] P. Bailis, E. Gan, S. Madden, D. Narayanan, K. Rong, and S. Suri. Macrobase: Analytic monitoring for the internet of things. *arXiv preprint arXiv:1603.00567*, 7, 2016.

[3] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. In *6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*.

[4] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: Online modelling and performance-aware systems. In *HotOS*, pages 85–90, 2003.

[5] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy. *Site Reliability Engineering: How Google Runs Production Systems.* " O'Reilly Media, Inc.", 2016.

[6] N. T. Blog. Lessons from Building Observability Tools at Netflix. Retrieved March 2020 from `https://netflixtechblog.com/lessons-from-building-observability-tools-at-netflix-7cfafed6ab17`.

[7] S. Flanders. OpenCensus and OpenTracing are now OpenTelemetry. Retrieved March 2020 from `https://omnition.io/blog/opencensus-and-opentracing-are-now-opentelemetry/`.

[8] S. Flanders and Y. Shkuro. A Picture is Worth a 1,000 Traces. Retrieved February 2020 from `https://www.shkuro.com/talks/2019-11-18-a-picture-is-worth-a-thousand-traces/`.

[9] R. Fonseca and J. Mace. We are Losing Track: a Case for Causal Metadata in Distributed Systems.

[10] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: A Pervasive Network Tracing Framework. In *4th USENIX Symposium on Networked Systems Design and Implementation (NSDI '07)*.

[11] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, et al. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems. In *24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*.

[12] Jaeger: Open Source, End-to-End Distributed Tracing. Retrieved June 2019 from `https://www.jaegertracing.io/`.

[13] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi, V. Vekataraman, K. Veeraraghavan, and Y. J. Song. Canopy: An End-to-End Performance Tracing And Analysis System. In *26th ACM Symposium on Operating Systems Principles (SOSP '17)*.

[14] S. Karumuri. Distributed Tracing at Pinterest with New Open-Source Tools. Retrieved March 2020 from `https://medium.com/pinterest-engineering/distributed-tracing-at-pinterest-with-new-open-source-tools-a4f8a5562f6b`.

[15] S. P. Kavulya, S. Daniels, K. Joshi, M. Hiltunen, R. Gandhi, and P. Narasimhan. Draco: Statistical diagnosis of chronic problems in large distributed systems. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12. IEEE, 2012.

[16] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[17] P. Las-Casas, G. Papakerashvili, V. Anand, and J. Mace. Sifter: Scalable Sampling for Distributed Traces, without Feature Engineering. In *11th ACM Symposium on Cloud Computing (SOCC '19)*.

[18] J. Mace and R. Fonseca. Universal Context Propagation for Distributed System Instrumentation. In *13th ACM European Conference on Computer Systems (EuroSys '18)*.

[19] J. Mace, R. Roelke, and R. Fonseca. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *25th ACM Symposium on Operating Systems Principles (SOSP '15)*.

[20] G. Mann, M. Sandler, D. Krushevskaja, S. Guha, and E. Even-Dar. Modeling the Parallel Execution of Black-Box Services. In *3rd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '11)*.

[21] T. Munzner. *Visualization Analysis and Design*. CRC Press, 2014.

[22] R. R. Sambasivan, I. Shafer, M. L. Mazurek, and G. R. Ganger. Visualizing Request-Flow Comparison to Aid Performance Diagnosis in Distributed Systems. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2466–2475, 2013.

[23] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger. Diagnosing Performance Changes by Comparing Request Flows. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11)*.

[24] 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. Technical Report, Google, 2010.

[25] C. Sridharan. Distributed Tracing: We've Been Doing It Wrong. Retrieved February 2020 from `https://medium.com/@copyconstruct/distributed-tracing-weve-been-doing-it-wrong-39fc92a857df`.

[26] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. R. Ganger. Stardust: Tracking Activity in a Distributed Storage System. In *2006 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '06)*.

[27] Zipkin. Retrieved July 2017 from `http://zipkin.io/`.

[28] C. Wang, I. A. Rayan, G. Eisenhauer, K. Schwan, V. Talwar, M. Wolf, and C. Huneycutt. Vscope: middleware for troubleshooting time-sensitive data center applications. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 121–141. Springer, 2012.

[29] Y. Wu, A. Chen, and L. T. X. Phan. Zeno: Diagnosing Performance Problems with Temporal Provenance. In *16th USENIX Conference on Networked Systems Design and Implementation (NSDI '19)*.