Snicket: Query-Driven Distributed Tracing

Jessica Berg[♦], Fabian Ruffy[♦], Khanh Nguyen[♦],
Nicholas Yang[♦], Taegyun Kim[♦], Anirudh Sivaraman[♦],
Ravi Netravali[♣], Srinivas Narayana[♠]
[♦] New York University, [♣] Princeton University, [♠] Rutgers University

ABSTRACT

Increasing application complexity has caused applications to be refactored into smaller components known as microservices that communicate with each other using RPCs. Distributed tracing has emerged as an important debugging tool for such microservice-based applications. Distributed tracing follows the journey of a user request from its starting point at the application's front-end, through RPC calls made by the front-end to different microservices recursively, all the way until a response is constructed and sent back to the user. To reduce storage costs, distributed tracing systems sample traces before collecting them for subsequent querying, affecting the accuracy of queries on the collected traces.

We propose an alternative system, Snicket, that tightly integrates querying and collection of traces. Snicket takes as input a database-style streaming query that expresses the analysis the developer wants to perform on the trace data. This query is compiled into a distributed collection of microservice extensions that run as "bumps-in-the-wire," intercepting RPC requests and responses as they flow into and out of microservices. This collection of extensions implements the query, performing early filtering and computation on the traces to reduce the amount of stored data in a query-specific manner. We show that Snicket is expressive in the queries it can support and can update queries fast enough for interactive use.

ACM Reference Format:
Jessica Berg, Fabian Ruffy, Khanh Nguyen, Nicholas Yang, Taegyun Kim, Anirudh Sivaraman, Ravi Netravali, Srinivas Narayana. 2021. Snicket: Query-Driven Distributed Tracing. . In *The Twentieth ACM Workshop on Hot Topics in Networks (HotNets '21), November 10–12, 2021, Virtual Event, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3484266.3487393

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HotNets'21, November 10-12, 2021, Virtual Event, UK
© 2021 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-9087-3/21/11...$15.00
https://doi.org/10.1145/3484266.3487393

1 INTRODUCTION

Growing application complexity has led organizations to decompose large web services into a collection of smaller components, known as microservices, that communicate with each other over an RPC interface [6]. When a user issues a request to a web service (e.g., for the landing page of a social network), the request is first received by a front-end microservice. The front-end microservice then issues RPCs to internal microservices, which in turn might call other microservices recursively to construct a response for the user.

Debugging such microservice-based applications is difficult because microservices are distributed across multiple compute nodes. An important debugging tool for such applications is distributed tracing [5, 23]. Distributed tracing tracks the flow of an incoming user request through the collection of traversed microservices and represents the request as a *trace*: a tree that captures parent-child relationships between all RPCs originated by a particular user request along with some metadata of each RPC (e.g., RPC latency). Distributed tracing systems [4, 8, 11, 17, 23] capture and persist traces in a database to permit subsequent querying by developers.

Storing a trace for every user request is prohibitively expensive. Hence, all existing tracing systems sample traces in some way. Two sampling strategies are commonly deployed today. Systems based on *head-based sampling* [4, 17, 23] sample user requests for tracing and storage before the requests spawn subsequent RPCs at the front-end. *Tail-based sampling* [13] approaches sample traces for storage after the request finishes execution and a full trace is available. Tail-based sampling enables decisions informed by trace contents, but is more complex than head-based sampling.

Unfortunately, the existing trace sampling approaches collect either more or less data than is actually needed to answer developer queries about traces. On the one hand, even if a developer is only interested in certain trace properties like end-to-end request latency, data is still persisted at the granularity of whole traces; this means more information is often

collected than needed for the query. On the other hand, uniform head-based sampling may miss anomalous traces, which are crucial to debugging, and tail-based sampling filters for specific types of traces, potentially missing traces relevant to subsequent queries. In either case, the data that is collected after sampling may not be what is required to answer the developer's query accurately.

Here, we take a different approach: *tightly coupling trace data collection and querying*. Unlike existing tracing systems, our output is not a database of traces to be queried. Rather the database is itself created by the developer's queries and captures precisely properties of traces that are of interest to the developer. We present a query system, Snicket, that takes a developer's queries as input, and produces a database populated by answers to those queries—no more and no less.

Snicket's input query language is database-style, high-level, and graph-centered. It allows the developer to work under the illusion that they can process every single trace in a centralized location in a streaming fashion to extract useful insights. The developer's query specifies what traces the developer is interested in (e.g., those with a particular error code), how to process these traces to extract useful information (e.g., end-to-end request latency), and how to aggregate multiple traces to produce useful summary statistics (e.g., mean end-to-end request latency across multiple traces). To get answers to the query, Snicket compiles the query to a distributed collection of *microservice extensions,* one per microservice. These extensions run as a bump-in-the-wire before and after the application logic within the microservice.

Our extension-based approach is enabled by two recent developments in microservices: (1) the emergence of service proxies and (2) support for programmability in these proxies through WebAssembly (WASM) [25] bytecode extensions. First, diverse microservices share common functions (e.g., authenticating users, granting those users different privileges, and load balancing across microservice replicas). Over time, such common functions have been factored out of the microservice's application logic and moved into a common infrastructure software layer known as the service proxy (e.g., Envoy and Linkerd [2, 3]). These service proxies effectively operate as application-layer switches and serve as the data plane of the inter-microservice network. Second, *bytecode extensions* are a new feature of service proxies [1] that allows them to be extended using WASM programs. These programs can be developed in a language that supports compilation to WASM such as C++ or Rust. Thus, these extensions augment service proxies with programmability, similar to programmable switches and network-interface cards. Snicket implements distributed tracing by compiling developer queries into bytecode extensions running within service proxies.

There are two key challenges in trying to compile queries on traces into service proxy extensions. First, at any time while a trace is being created, no extension has a full view of all microservices or of the trace itself. Yet some computation must be done at that time in order to integrate data collection and querying. Allowing the developer the illusion of having a full view of both, while individual microservices do not, is challenging. §3.2 discusses how the Snicket compiler handles this.

Second, it is important to ensure service proxy extensions do not add untenable overhead (CPU usage, latency, etc). The extensions run as a bump-in-the-wire, meaning any latency they incur will have a direct effect on the performance of the application. Because many microservice applications run in the cloud, extra CPU usage implies more money spent. §5 discusses potential solutions to reduce this overhead.

In preliminary evaluations of Snicket, we test Snicket on an open-source microservice benchmark called Online Boutique [7]. We find that Snicket adds modest latency (~17ms) and CPU overhead (9% increase). We also evaluate Snicket's expressiveness and how quickly Snicket's queries can be updated. Snicket is currently available at https://github.com/dyn-tracing/snicket_compiler.

2 BACKGROUND AND RELATED WORK

Distributing Tracing. Distributed tracing is the practice of tracking a user's request from its entry to its exit. Each RPC, from a parent to a child in the trace, is captured as a *span*. The span may also contain metadata about that RPC such as the latency of the RPC. *Baggage* is the data that is propagated within RPCs across multiple microservices in order to collect information about the trace. All spans issued as a result of the same user request can be assembled into a trace: a directed tree where edges represent caller-callee relationships.

An Ideal System. To contextualize Snicket's tradeoffs, we consider an idealized tracing system that records every span, and sends them to a centralized service to be stored forever. That system would incur a small but significant overhead on the application and would have perfect visibility into any point in the past, but would have prohibitively large storage costs. All current tracing systems reduce these storage costs in some way, and in doing so increase the overhead on the application and/or reduce visibility into data.

Trace Database Systems. Most tracing systems address the tradeoff by sampling: they store only a fraction of the traces in a database for later querying. Dapper, Jaeger, and Canopy employ uniform head-based sampling, which has the advantage of simplicity, but may miss important unusual traces, thus restricting visibility [4, 11, 23]. Canopy also employs a form of developer-defined tail-based sampling: based on developer input, Canopy decides which traces to keep for later querying [11]. Lightstep uses dynamic sampling, which

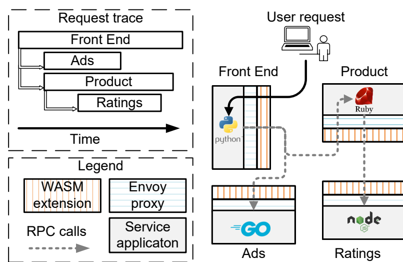

Figure 1: An example microservice application. Snicket's generated code runs in the Envoy proxy as extensions.

is similar to tail-based sampling, but instead of considering traces one at a time, it considers all traces from the last one hour to decide what to store [26]. LightStep's dynamic tracing prioritizes unusual traces within the last one hour of traces.

In all these systems, the trace data is first sampled in a system-specific way and then persisted to long-term storage where it may be queried. Because querying happens on the database after sampling, query results may not be representative of the full stream of traces observed by the microservices. In contrast, Snicket's query results *are* accurate because Snicket logically operates on *all* traces seen by the application. However, only the results of Snicket's query are stored.

Query-Based Systems. Pivot Tracing takes a similar approach to Snicket by tightly tying collection and querying together [15]. Pivot Tracing takes a query as input, and compiles this query down to dynamic tracepoints throughout a distributed system that find the query answer. Similar to Snicket, it stores only the query results, not the traces from which they are derived. Unlike Snicket, Pivot Tracing does not support graph-based queries. It also requires more intrusive changes to the application for deployment, which Snicket sidesteps by operating at the proxy extension layer.

Recently, there has been some interest in *offline* graph analytics of traces [8, 14] because traces are graph structures that are well suited to graph-processing systems. In these systems, a trace is treated as a graph and not as a flat collection of spans; so events from different branches of the same trace are more easily correlated and processed. Snicket captures similar graph-based information, while tightly binding together collection and querying in an *online* streaming manner.

3 DESIGN

3.1 Input: Query Language

In Snicket, a trace is modeled as a directed tree rooted at the front-end. Each vertex is a unique visit to a microservice, and each edge is an RPC. Snicket's query language syntax is based on OpenCypher [16] (Figure 2). It allows a microservice developer to specify both the graph structure and attributes of traces they are interested in. The input to a query is the stream of traces created by developer requests, and the output is the answer to the query, either expressed as a single value per trace or as the result of an aggregation function over the per-trace values. The Snicket compiler compiles queries into a collection of WASM extensions running in the Envoy proxy (Figure 1). Snicket's language constructs are described below.

Structural Patterns Using `MATCH`. `MATCH` corresponds to a structural filter, which specifies the structure of the graph to match on. For example, one could ask for a trace containing a subtree with a parent and 5 children, as shown in Figure 3.

Attribute Specification Using `WHERE`. `WHERE` corresponds to an attribute filter; it specifies any vertex-level attributes the vertices referenced in the structural filter might have (e.g., vertex "a" must be named "shoppingcart-service"). It also can specify trace-level attributes (e.g., the latency of the entire trace or the trace ID). *Inherent* vertex-level attributes come directly from Envoy's interface to WASM. This allows the language to grow as Envoy expands what it makes available to WASM. These attributes are either inherent to the vertex's microservice (e.g., the microservice's name) or the RPC response sent back by the vertex to its parent (e.g., the size of the response or a header within the response). Thus, Snicket developers also have access to whatever application-specific information is sent through RPC response headers.

Developer-Defined Attributes with `function(input)`. This construct allows new attributes to be created from inherent vertex and trace-level attributes. Inherent attributes that are built into the Envoy service proxy are automatically available and accessible through dot notation (e.g., `vertex.name`). A developer can define a mapping function on these inherent attributes to create new attributes, like the height of the trace graph. They are recursively defined: the root's value will be considered the attribute for the trace as a whole. For example, if a developer wanted the height of a tree, they could recursively define it to be the maximum height of a vertex's children, plus one. Then, the root vertex's height would also be the trace's height.

Query Answers with `RETURN`. `RETURN` can either return one value per trace, (e.g., `RETURN latency (trace)`) or an aggregation (e.g., `RETURN avg (latency (trace))`) over attributes across traces. In the first case, the output of the query is a single value per trace (e.g., a vertex's name for every trace that matches the structural and attribute filters). In the second case, the output of the query is the result of the aggregation function implemented on the returned element (e.g., the average latency of multiple traces). The aggregation functions may maintain arbitrary intermediate data to arrive at their final value. For example, for an average, the aggregation

Operator	Example Expression in OpenCypher	Description
Structural Filter	MATCH (a) -> (b)	Defines an arbitrary graph structure to match on.
Attribute Filter	WHERE a.response.total_size = 500 AND height(trace) = 5	Filters based on attributes of vertices and traces.
Developer-Defined Attributes	latency(b)	Creates a new, developer-defined attribute for vertices and traces.
Return	RETURN latency(b)	Defines what information will be returned to the developer.
Aggregate	RETURN average(latency(b))	Specifies an aggregation function that should be applied to data across traces before returning to the developer.

Figure 2: A table demonstrating each language construct in Snicket, an example corresponding to that construct in OpenCypher syntax, and a description of the construct.

function implementation keeps a running tally of the total sum, and the number of instances seen. When a new value is given to the aggregation function because another trace has been completed, the aggregation function updates its two internal values, and divides them to get the value to be placed in storage. Aggregation functions are developer-defined.

Example Scenario. As an example of Snicket in practice, consider the scenario where a company is switching from local to cloud-based machines. In the midst of the transition, queries are being load balanced across multiple replicas, some local, and some on the cloud. A developer notices that many slow requests go through these replicas. A reasonable hypothesis is that there may be some difference between how the local and cloud replicas are set up. However, that is only one possible explanation of many. To test that explanation, the developer formulates the query

```
MATCH (a)->(b)
WHERE
     a.vertex.workload.SERVICE_NAME
         == "frontend"
     AND a.downstream.address
         == local_address
RETURN avg(latency(trace))
```

In storage, the developer will be able to access a continually updated average of the latency of traces that went through the local replicas. If this number is normal, there is likely a problem on the cloud replicas. If not, then the developer can change the query's IP address to the cloud IP, and figure out if the local replicas are the problem.

3.2 Output: WASM Filters

Snicket's compiler generates WASM extensions from input queries. WASM runs in a safe, memory-sandboxed environment that is isolated from the service proxy [25]. The WASM environment has a well-defined interface with the Envoy service proxies; it can put data into proxy storage, inspect incoming and outgoing messages, and learn about the status and placement of the service proxy. The WASM environment also has access to Envoy-supplied attributes like the trace ID. There is one service proxy per microservice. For simplicity in compiler implementation, the compiler currently generates the same extension for each proxy except for the storage, which is a container managed by Snicket to keep the results of a query. When the developer wants to know the results of a query, they query the storage container.

3.3 The Snicket Compiler

We now describe how the compiler implements each language construct: the attribute filter, structural filter, developer-defined attributes, return, and aggregation. The compiler creates as output two extensions: the main extension that runs on all application microservices, and an aggregation extension which runs only on the storage container. Both extensions run in the proxies, so no instrumentation is needed in the application as long as it uses any service proxy with an extension mechanism. Our implementation uses the Envoy service proxy and WASM bytecode extensions.

Attribute Collection. First, as requests pass through various microservices, relevant Envoy attributes are added to the baggage: the data that is propagated alongside an RPC [18] as it hops across microservices to complete the user request. This happens within the main extension. The attributes are accessible through the Envoy interface, and will later determine whether the trace graph matches the query's filters.

Attribute and Structural Filtering. As RPC calls are made in response to incoming user requests, the WASM extension creates a tree of RPCs in an online fashion, starting at the leaves. This tree is part of the baggage that is propagated between microservices. Once a response for an RPC is processed, the responding microservice is added to the tree as a vertex—effectively, a post-order traversal of the trace. As the tree is created from leaves to root, an *isomorphism algorithm* [22] is run in a distributed manner at each microservice, finding matches to the structural and attribute patterns specified in the query, using only the information collected thus far. The algorithm executes both structural and attribute filters at the same time.

Developer-Defined Attributes with `function(input)`. Developers can also create new recursively defined attributes. The functions defining these attributes execute once for each vertex: when a response is received from a callee vertex and the caller vertex becomes part of the trace, then the caller vertex will define the attribute for itself. Then the attribute is added to baggage in the same way as inherent attributes. To create a new attribute, the developer refers to an attribute using parenthesis notation (e.g., `height(trace)`) within the query, and provides the function defining the attribute as part of the query ingested by the compiler.

Return. In the main extension, once it has been determined that the attribute and structural filters are satisfied, the extension sends the result to the storage container. The result is determined by the return statement and sent to the storage container as a (Trace ID, value) pair, in order to distinguish which result came from which trace.

Aggregate. The developer can also define their own functions in the aggregation construct. A developer can create an aggregation function that takes in anything that can be put as a `RETURN` value, and continuously outputs one entry to be put into the storage container. Thus the developer can do some summary computation on return values, and store these summaries, instead of storing per-trace values.

In the example scenario in section 3.1, the input to the aggregation function was `latency (trace)` and the aggregation function is `avg`. The main extension sends (Trace ID, Latency) pairs to the storage container. But rather than immediately sending these pairs to storage, the aggregation extension on the storage container intercepts these pairs and continuously recomputes a running average of latencies across all pairs seen so far. This running average is maintained within the storage container and updated with every new pair.

4 EVALUATING SNICKET

4.1 Language Expressiveness

In Figure 3, we show various examples of Snicket queries. These queries could include correctness checks, investigation of anomalous data emitted by the application, or debugging of erroneous user requests. Two defining features of the Snicket's language that are difficult to express with prior systems are: (1) the ability to match on specific graph structures and (2) the ability to create new developer-defined attributes *without* restarting applications.

4.2 Interactivity

How long does it take until Snicket's extensions take effect? The accepted method up until 2020 for replacing a WASM extension was through uploading the extension to a Kubernetes *config map* [21]. However, Kubernetes also makes this config map impossible to overwrite [24]. Hence, every time the contents of the config map (our Snicket extension) are overwritten, the entire service proxy must be restarted. Although this does not affect application functionality, it is slow. Recently, a new way to refresh WASM extensions has been developed [10]. It is still an experimental feature, and has limited uses. Using the new method, we improved the extension refresh time from 30–103 seconds to 0.6–0.9 seconds. We hope that as the feature moves beyond experimental, Snicket can become more interactive.

4.3 Cost and Performance

We measure the cost and performance of Snicket by two metrics: (1) the extra compute costs that needs to be paid to run Snicket alongside an application, and (2) the latency added by Snicket to user requests.

Application. We use Online Boutique, a microservices application of 10 microservices [7]. The only change we made to the application itself is increasing the requested CPU for each pod to 601 millicores (1 millicore is one thousandth of a core); this allows for better performance of the base application.

Cluster Configuration. The cluster is initially given 7 nodes of machine type "e2-highmem-4" (4vCPU, 32GB memory) on the Google Cloud Platform. Horizontal autoscaling is enabled with a threshold of 40 percent CPU utilization [19], and all pods are allowed at most 10 replicas, with the exception of the frontend, which is allowed 30 replicas. None of the experiments reached the autoscaler limits. We enabled the default Kubernetes cluster autoscaler [12].

Load Generator. We use a load generator, Locust [9], to create load on the application. Locust spawns five users every second until it reaches 500 users. The load generator was located in a VM in the same cloud zone as the application. Each user sends a request, waits a random period between 1 and 3 seconds, then sends another request, and so on. During this time, both the horizontal and cluster autoscalers are allowed to stabilize to handle the load. Then, we record latency measurements for 400 seconds and record the extra resources used to run the application. We measure three cases: (1) the application alone, (2) the application with a no-op WASM extension running in each service proxy; this is meant to capture how much overhead is being added by deploying WASM extensions at all, (3) the application with extensions generated by Snicket. We use the query:

```
MATCH (a)-[]->(b)-[]->(c)
WHERE c.node.metadata.WORKLOAD_NAME='ratings-v1'
RETURN a.node.metadata.WORKLOAD_NAME
```

Results. Without Snicket running, the load forces the autoscaler to create 11 nodes, and the latency has a median of

Question	Query	Response in Storage
Which services are making calls to the cache?	MATCH (a) -[]-> (b) WHERE b.node.metadata.WORKLOAD_NAME == cache RETURN a.node.metadata.WORKLOAD_NAME	A list of all services that called the cache
How long is latency in a 5 child wide graph?	MATCH (a) -[]-> (b), (a) -[]-> (c), (a) -[]-> (d), (a) -[]-> (e), (a) -[]-> (f) RETURN latency(trace)	A list of latencies of traces with >= 5 children
How many services do requests going through a, b, and c normally go through?	MATCH (a) - []-> (b) -[]-> (c) WHERE a.node.metadata.WORKLOAD_NAME == frontend AND b.node.metadata.WORKLOAD_NAME == productservice AND c.node.metadata.WORKLOAD_NAME == currencyservice RETURN median(height(a))	The median of the height of the subtree rooted at frontend that goes through the product and currency services
Is the header "foo" present in the following trace graph?	MATCH (a) -> (b), (a) -> (c), (b) -> (d), (d) -> (e) RETURN foo(a)	A list of each trace ID mapped to its foo header

Figure 3: Examples of Snicket queries.

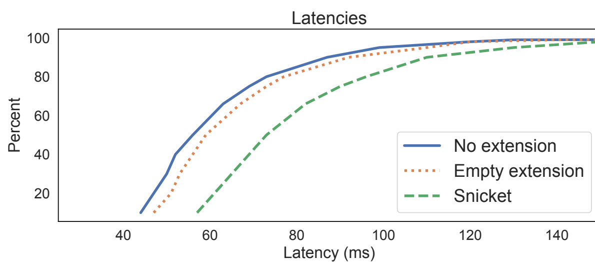

Figure 4: A CDF of latencies comparing no WASM extension at all, the no-op WASM extension, and the WASM extension generated by Snicket.

56 ms (95th percentile: 99 ms). With a no-op extension running, the autoscaler creates 11 nodes, and the latency has a median of 59 ms (95th percentile: 110 ms). In other words, the effect of simply including a WASM extension at all is relatively low. However, an extension generated from an example query forces the autoscaler to create 12 nodes and increases median latency to 73 ms (95th percentile: 130 ms).

Comparison to other systems. To contextualize this result, it is useful to look at other systems. Dapper, without sampling, has 16 percent latency overhead [23]. Snicket, which pushes computation to the proxies, has about 30% latency overhead (56 ms to 73 ms). While this is a significant increase over Dapper, we believe it is not insurmountably large (§5).

5 FUTURE WORK

Snicket only persists the results of queries on traces in long-term storage—as opposed to the traces themselves. Hence, the data persisted by Snicket in long-term storage is tied closely to the query issued to Snicket. This limits Snicket's *historical visibility*: if a developer wants to reanalyze historic data with a new query, the developer is limited by the queries used at the time the data was collected. Ideally, the developer should be able to query both current and historic traces with the illusion that all information is always available.

To do so, we are exploring improvements to Snicket that allow us to compactly persist all the information from a trace by exploiting correlations between and across traces to achieve lossless compression of the traces. We also plan to organize this data in a manner that facilitates future queries on historic and current trace data. For this, we will develop indexing mechanisms to easily access information belonging to a particular trace, microservice, or time window. These improvements will augment the *online* querying abilities of Snicket with the ability to also access historic traces in queries.

We are also looking into optimizations to reduce the 30% latency overhead of Snicket. Because we have measured a version of the application with the empty WASM extension, we know that the overhead to get into and out of the WASM runtime is relatively low (~3 ms) (Figure 4). Hence, the bulk of the latency overhead is imposed by Snicket's autogenerated extensions, and a promising path for optimization is through optimizing the code Snicket generates. We are currently investigating moving more of the extension computation out of the critical path of requests so that minimal overhead is added by the WASM extensions. We are also considering optimizations such as superoptimization [20] of WASM bytecode to produce bytecode that adds low overhead to RPC processing.

6 CONCLUSION

We have presented Snicket, a system for query-guided distributed tracing that compiles developer queries on traces into a distributed collection of extensions. In preliminary experiments, Snicket incurs about 30% latency overhead relative to an application without tracing. We are continuing to develop Snicket to reduce its current sources of overhead and improve its historic visibility into traces.

Acknowledgements. We thank the reviewers for their insightful comments. This research was partially supported by NSF grants CNS-2152313, CNS-1901510, and CNS-2008048.

REFERENCES

[1] Craig Box, Mandar Jog, Plevyak John, Ryan Louis, Sikora Piotr, Kohavi Yuval, and Weiss Scott. 2020. Redefining Extensibility in Proxies - Introducing WebAssembly to Envoy and Istio. https://istio.io/latest/blog/2020/wasm-announce/. Accessed: 2021-06-25.

[2] Cloud Native Computing Foundation. 2021. Envoy: an open source edge and service proxy, designed for cloud-native applications. https://www.envoyproxy.io/. Accessed: 2021-06-25.

[3] Cloud Native Computing Foundation. 2021. Linkerd: The world's lightest, fastest service mesh. https://linkerd.io/. Accessed: 2021-06-25.

[4] Cloud Native Computing Foundation. 2021. Open Source, End-to-End Distributed Tracing. https://www.jaegertracing.io/. Accessed: 2021-06-25.

[5] Rodrigo Fonseca, George Porter, Randy H. Katz, and Scott Shenker. 2007. X-Trace: A Pervasive Network Tracing Framework. In *USENIX NSDI*.

[6] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *ACM ASPLOS*.

[7] Google, Inc. 2021. Online Boutique: a Cloud-Native Microservices Demo Application. https://github.com/GoogleCloudPlatform/microservices-demo/. Accessed: 2021-06-25.

[8] Xiaofeng Guo, Xin Peng, Hanzhang Wang, Wanxue Li, Huai Jiang, Dan Ding, Tao Xie, and Liangfei Su. 2020. Graph-Based Trace Analysis for Microservice Architecture Understanding and Problem Diagnosis. In *ACM ESEC/FSE*.

[9] Jonatan Heyman, Joakim Hamrén, Carl Byström, and Hugo Heyman. 2021. Locust: An Open Source Load Testing Tool. https://locust.io/. Accessed: 2021-06-25.

[10] Istio. 2021. Distributing WebAssembly Modules (Experimental). https://istio.io/latest/docs/ops/configuration/extensibility/wasm-module-distribution/. Accessed: 2021-06-25.

[11] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Vekataraman, Kaushik Veeraraghavan, and Yee Jiun Song. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. In *ACM SOSP*.

[12] Kubernetes. 2021. kubernetes/autoscaler: Autoscaling components for Kubernetes. https://github.com/kubernetes/autoscaler. Accessed: 2021-10-11.

[13] Pedro Las-Casas, Giorgi Papakerashvili, Vaastav Anand, and Jonathan Mace. 2019. Sifter: Scalable Sampling for Distributed Traces, without Feature Engineering. In *ACM SOCC*.

[14] Pavol Loffay. 2020. Data analytics with Jaeger aka Traces Tell Us More! https://medium.com/jaegertracing/data-analytics-with-jaeger-aka-traces-tell-us-more-973669e6f848. Accessed: 2021-06-25.

[15] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *ACM SOSP*.

[16] Neo4j, Inc. 2019. *Cypher Query Language Reference, Version 9*.

[17] OpenZipkin. 2021. Zipkin. https://zipkin.io/. Accessed: 2021-06-25.

[18] Austin Parker, Daniel Spoonhower, Jonathan Mace, Ben Sigelman, and Rebecca Isaacs. 2020. *Distributed Tracing In Practice: Instrumenting, Analyzing, and Debugging*. O'Reilly Media.

[19] Google Cloud Platform. 2021. Horizontal Pod autoscaling | Kubernetes Engine Documentation. https://cloud.google.com/kubernetes-engine/docs/concepts/horizontalpodautoscaler.

[20] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2012. Stochastic Superoptimization. Accessed: 2021-10-11.

[21] Toader Sebastian. 2020. How to Write WASM Filters for Envoy and Deploy It With Istio. https://banzaicloud.com/blog/envoy-wasm-filter/#create-a-config-map-to-hold-the-wasm-binary. Accessed: 2021-06-25.

[22] Ron Shamir and Dekel Tsur. 1999. Faster Subtree Isomorphism. *Journal of Algorithms* (1999).

[23] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Technical Report. Google, Inc.

[24] Joel Smith. 2018. Ensure That the Runtime Mounts RO Volumes Read-Only by. https://github.com/kubernetes/kubernetes/pull/58720. Accessed: 2021-06-25.

[25] W3C Working Group. 2021. WebAssembly: a Binary Instruction Format for a Stack-Based Virtual Machine. https://webassembly.org/. Accessed: 2021-06-25.

[26] Robin Whitmore. 2021. How Lightstep Works. https://docs.lightstep.com/docs/how-lightstep-works. Accessed: 2021-06-25.