Tracezip: Efficient Distributed Tracing via Trace Compression

ZHUANGBIN CHEN, School of Software Engineering, Sun Yat-sen University, China
JUNSONG PU, Beijing University of Posts and Telecommunication, China
ZIBIN ZHENG*, School of Software Engineering, Sun Yat-sen University, China

Distributed tracing serves as a fundamental building block in the monitoring and testing of cloud service systems. To reduce computational and storage overheads, the *de facto* practice is to capture fewer traces via sampling. However, existing work faces a trade-off between the completeness of tracing and system overhead. On one hand, *head-based sampling* indiscriminately selects requests to trace when they enter the system, which may miss critical events. On the other hand, *tail-based sampling* first captures all requests and then selectively persists the edge-case traces, which entails the overheads related to trace collection and ingestion. Taking a different path, we propose Tracezip in this paper to enhance the efficiency of distributed tracing via *trace compression*. Our key insight is that there exists significant redundancy among traces, which results in repetitive transmission of identical data between services and the backend. We design a new data structure named Span Retrieval Tree (SRT) that continuously encapsulates such redundancy at the service side and transforms trace spans into a lightweight form. At the backend, the complete traces can be seamlessly reconstructed by retrieving the common data that are already delivered by previous spans. Tracezip includes a series of strategies to optimize the structure of SRT and a differential update mechanism to efficiently synchronize SRT between services and the backend. Our evaluation on microservices benchmarks, popular cloud service systems, and production trace data demonstrates that Tracezip can achieve substantial performance gains in trace collection with negligible overhead. We have implemented Tracezip inside the OpenTelemetry Collector, making it compatible with existing tracing APIs.

CCS Concepts: • **Computer systems organization → Cloud computing**; **Reliability**; **Maintainability and maintenance**; **Redundancy**; • **Software and its engineering → Software maintenance tools**.

Additional Key Words and Phrases: Distributed tracing, Trace compression, Cloud computing, System monitoring

ACM Reference Format:
Zhuangbin Chen, Junsong Pu, and Zibin Zheng. 2025. Tracezip: Efficient Distributed Tracing via Trace Compression. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA019 (July 2025), 23 pages. https://doi.org/10.1145/3728888

1 Introduction

In modern cloud systems, the adoption of loosely coupled designs for applications and services has marked a significant paradigm shift in software architecture. While such modular design brings the benefits of scalability and operational flexibility, it also complicates different aspects of software development and maintenance, including testing, debugging and diagnosing. This can be

*Zibin Zheng is the corresponding author.

Authors' Contact Information: Zhuangbin Chen, School of Software Engineering, Sun Yat-sen University, Zhuhai, China, chenzhb36@mail.sysu.edu.cn; Junsong Pu, Beijing University of Posts and Telecommunication, Beijing, China, angrychow@bupt.edu.cn; Zibin Zheng, School of Software Engineering, Sun Yat-sen University, Zhuhai, China, zhzibin@mail.sysu.edu.cn.

ACM 2994-970X/2025/7-ARTISSTA019
https://doi.org/10.1145/3728888

largely attributed to the cascading effect of failures [5, 9], i.e., a single failure in one service can quickly propagate to other interconnected services and components. As such, distributed tracing has emerged as an essential reliability management solution in cloud systems. This is primarily due to its ability to provide a comprehensive and granular view of the interactions between services, allowing for the precise identification of where and how failures occur and spread.

In cloud service systems, unusual and edge-case system behaviors, such as tail latency and resource contention, are rare by definition. To achieve high coverage of outlier system events, it is necessary to trace *all* requests. In production environments, this may result in substantial trace data, incurring significant overhead and costs related to trace generation, collection, and ingestion. To mitigate this problem, various *trace sampling* techniques have been proposed, which can be categorized into two main types, i.e., *head-based sampling* [48] and *tail-based sampling* [4, 14, 17, 48]. Head-based sampling uniformly collects traces at random based on a small sampling rate (e.g., 1% [20, 22, 48]). The sampling decision is made before request execution, and only the sampled requests will be traced. On the other hand, tail-based sampling captures traces for all requests, and decides whether to retain a trace based on the execution details such as latency and HTTP status code. In this regard, many machine learning techniques have been applied to automatically select informative and uncommon traces. However, given the inherent unpredictability of the true value of traces, trace sampling struggles to capture the full spectrum of critical information necessary for effective software testing and failure diagnosis.

In this paper, we propose Tracezip, an online and scalable solution to address the overhead of distributed tracing by *trace compression*. Tracezip transforms spans into a concise representation upon their generation at the service side, which can be seamlessly decompressed at the backend side. This strategy significantly reduces the volume of data that needs to be transmitted. We discuss two possible solutions for this end. The first is *offline log compression* [24, 37, 42, 43], which condenses log data after it has been aggregated at the backend for long-term persistence. As the primary goal is to save storage space, this approach often involves sophisticated algorithms which fail to meet the real-time requirements for trace collection. Moreover, it often requires processing the entire log dataset to achieve the optimal performance. In our scenario, online learning capability is crucial as traces are continuously generated in a stream. *General-purpose compression algorithms* (e.g., gzip and bzip) may also seem an out-of-box solution. However, they are designed for encoding arbitrary binary sequences, which can only exploit redundant information within a short sliding window (e.g., 32KB in gzip's Deflate algorithm).

To pursue more effective compression, Tracezip harnesses the global redundancy inherent in the structures of trace spans. Specifically, we design a new data structure, *Span Retrieval Tree* (SRT), based on the principles of prefix trees, which is able to continuously extract the set of key-value pairs commonly shared across spans. The SRT, when synchronized between services and the backend, serves as a reference mechanism to retrieve the identical data that have been previously transmitted by other spans. To manage computational and space complexity, Tracezip constantly restructures SRT into its most compact form and employs mapping techniques to further reduce its size. We also propose a differential update mechanism to effectively synchronize SRT between services and the backend. Tracezip is orthogonal to trace sampling methods and can work with log compression techniques once trace data have been efficiently transmitted to the backend.

We have implemented Tracezip inside OpenTelemetry Collector [34], one of the most popular tracing frameworks, making it compatible with existing tracing APIs. We deploy Tracezip to collect traces for an open-source microservices benchmark (i.e., Train Ticket [52]) and six application backend components in cloud environments (i.e., gRPC, Apache Kafka, Servlet, MySQL, Redis, MongoDB). We also evaluate the compression performance of Tracezip on Alibaba production traces [28, 41]. The experimental results show that Tracezip offers around 10%~45% performance

gain when working in conjunction with traditional compression schemes, i.e., gzip, bzip2, lzma. Moreover, TraceZip demonstrates negligible space overhead (i.e., several megabytes) and high efficiency.

The major contributions of this work are as follows:

- We propose TraceZip, the *first* online trace compression system by leveraging the inherent redundancy in trace span data. This information is captured by a new data structure we developed, the Span Retrieval Tree (SRT). By sharing SRT between services and the tracing backend, we can eliminate the redundancy associated with the repetitive transmission of identical data across multiple spans.
- We have implemented TraceZip inside OpenTelemetry Collector with a series of optimization strategies for SRT restructuring and synchronization. Experiments on both open-source systems and production trace dataset demonstrate that TraceZip can effectively compress traces with MB-scale space overhead and superior efficiency. The implementation and data of TraceZip are publicly available*.

The remainder of the paper is organized as follows. Section 2 introduces the background of distributed tracing and the motivation of this work. Section 3 and 4 describes the proposed methodology and system implementation. Section 5 presents the experiments and results. Section 6 discusses the related work. Finally, Section 7 concludes this work.

2 Background and Motivation

2.1 Distributed Tracing

Distributed tracing provides a detailed end-to-end view of requests as they traverse interconnected and multi-tier cloud service systems. The building blocks of a trace are called *spans*. They represent the individual slices of work performed across different machines and components that are visited by the request. Each span encapsulates various attributes, including *span name*, *parent span ID*, *span ID*, *start/end timestamps*, *events*, etc. This information enables the correlation and analysis of the request's lifecycle.

Figure 1 illustrates a typical procedure of tracking requests [30, 48] in modern distributed tracing frameworks, such as Jaeger [19] and Zipkin [54]. Upon the arrival of a new request to the application, the tracing framework assigns it a unique `trace_id` (①). However, not all requests will actually be traced, which is indicated by a flag, `sampled`. Both `trace_id` and `sampled` will then be propagated along the request at the application level, which is important for the completeness and coherence of a trace. If `sampled` is set, each component (e.g., a microservice instance) that handles the request will generate trace data (②), i.e., a span, using the tracing framework's client library (e.g., OpenTelemetry). The places that emit spans are called a *tracepoint*, and there could be multiple tracepoints serving different requests or different operations within a single request. The framework's client library then enqueues, serializes, and transmits trace spans (③) to its centralized backend collection infrastructure, or simply backend. The backend is responsible for continuously receiving (④), processing (⑤), and storing (⑥) trace data. Based on the `trace_id`, `parent_span_id`, and `span_id`, the backend can assemble spans that were dispersed across different components into a single coherent trace.

Given the details provided by these spans, operators can closely monitor and understand the impact that one component may have on the others. This makes trace data particularly useful for troubleshooting cross-component problems in large distributed systems. However, traces can be produced at high volume, incurring significant network, compute, and storage costs. In production scenarios, Google is estimated to generate approximately 1,000 TB of raw traces on a daily basis.

*https://github.com/OpsPAI/TraceZip

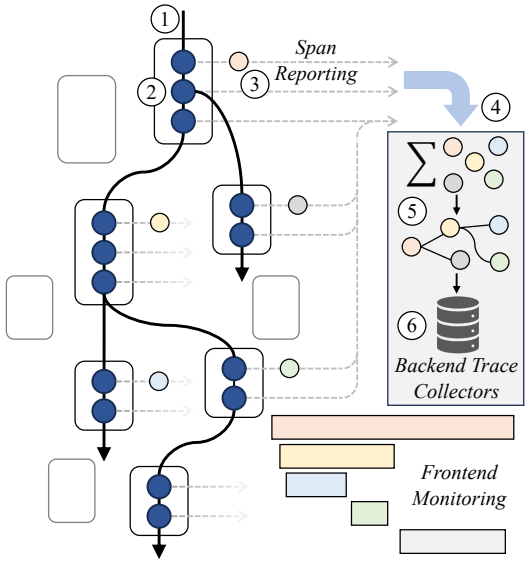

Fig. 1. A Typical Procedure of Distributed Tracing

Netflix needs to manage more than 2 billions of daily requests. To mitigate overheads, existing tracing frameworks often apply head-based sampling to trace only a small fraction of requests by randomly setting the `sampled` flag (①). This will inevitably increases the risk of overlooking system edge-case behaviors. On the other hand, tail-based sampling utilizes a filtering strategy, i.e., only persisting traces that exhibit outliers symptoms (⑥), e.g., high tail latency, error codes. However, tail-based sampling entails enormous costs, as it must trace all requests and ingest the trace data to make sampling decisions.

In this work, we take an orthogonal path to address the overhead challenges. We alleviate the lossy nature of sampling-based schemes by tracing more requests, yet without increasing the transmission overhead. Such a design can enhance the monitoring capability of edge-case behaviors. Our core idea lies in the observation that there exists a large amount of redundancy in trace data. Specifically, for each tracepoint, the generated spans may have repeated attribute values and events. This renders repetitive transmissions of identical trace data, and we see this could be an opportunity to improve the tracing efficiency. Recognizing this, we conduct a study to examine the redundancy inherent in trace data. The insights obtained serve as essential design principles of our approach.

2.2 A Study of the Redundancy in Trace Data

In this section, we present our study regarding the *redundancy* of trace data. By redundancy, we mean the recurrence of identical information, e.g., attributes and events, that is observed repeatedly across multiple traces. The identification of such repetitive patterns presents an opportunity to enhance the efficiency of distributed tracing through the application of compression techniques. By strategically reducing the redundancy, we can optimize the transmission overhead of trace data, thereby improving the scalability of the tracing infrastructure without compromising data integrity.

The studied systems include an open-source microservices benchmark, i.e., Train Ticket [52], which is popular in cloud-related research fields, and six application backend components that are selected for their widespread use in modern cloud systems, i.e., gRPC (Client and Server), Apache Kafka (Producer and Consumer), Servlet, MySQL, Redis, and MongoDB. They cover a diverse

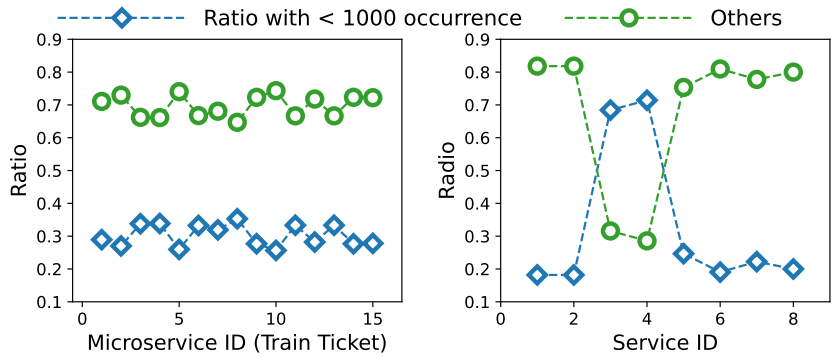

Fig. 2. Trace Data Redundancy Analysis

functionalities including message queuing, HTTP communication, remote procedure calls, database management, etc. To collect their traces, we leverage the zero-code instrumentation capabiltiy of OpenTelemetry [32]. It allows the collection of observability data, i.e., logs, metrics, and traces, for applications without the need to modify the source code. This is achieved by using libraries, plugins, or agent to instrument the libraries used by applications. It supports many programming languages (e.g., Java, Go, Python) and a wide range of popular libraries and frameworks, including requests and responses, database calls, message queue calls. By injecting typical workloads to the studied systems (Section 5.1.1), we collect more than 40GB of trace data in total.

We quantify trace redundancy by measuring the proportion of duplicate key-value (KV) pairs generated by each service. As a KV pair represents the fundamental unit of information within a trace, assessing its repetition enables us to gauge the degree of information overlap. Specifically, for each KV pair, we first count the number of its occurrence and then calculate its ratio over the total number of KV pairs in the dataset. For example, in a trace dataset containing 100 KV pairs, if two specific pairs appear once and ten times respectively, their redundancy ratios would be 1% and 10%, respectively. We calculate the fraction of KV pairs within each service which occur less than 1,000 times, as well as those exceeding this threshold, as shown in Figure 2. We make the following two important observations.

Traces are highly redundant. Based on the figures, we can see that for different microservices in Train Ticket, around 70% of the total KV pairs are highly repetitive. Similar situations can be found in the application components. The only exception is Apache Kafka, whose KV pairs tend to be unique. The reason behind is that Kafka's traces include the data from its message queues, increasing the randomness of its KV pairs. We will discuss its impact on compression during system evaluation (Section 5). The results indicate that a significant portion of the trace data is characterized by redundant information. An important reason is that services often engage in standard interactions and perform routine operations that generate trace data with similar patterns. By capturing the redundancy upon the generation of spans at the service side (①), we can preemptively eliminate the transmission of repetitive data that already exist in the backend. The backend can easily reconstruct the complete spans based on their redundancy patterns (⑥).

There exists structural redundancy among attributes. We also observe that there exists certain redundancy at span level. OpenTelemetry's semantic conventions [33] offer standardized guidelines for naming common attributes across different kinds of operations and data. This is essential for maintaining the uniformity and compatibility of naming scheme across languages, libraries,

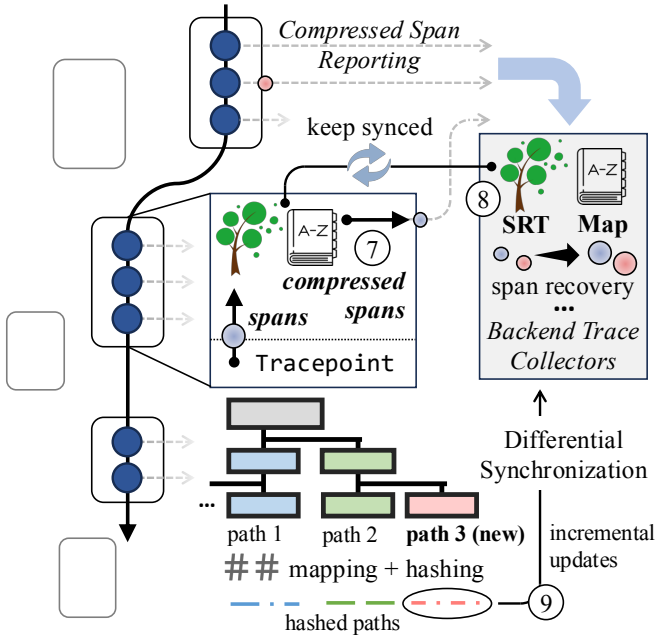

Fig. 3. System Architecture of Tracezip

and platforms. In the naming scheme, attribute names are organized into hierarchical namespaces to indicate their context or category, e.g., `network.local.address`, `network.local.port`, `network.peer.address`. We can see that they share some common words. The attribute values also exhibit similar commonality, e.g., Java Exception has `java.io.IIOException`, `java.io.EOFException`. By removing such fine-grained redundancy, we can further reduce the overheads of span transmission.

3 Methodology

3.1 Overview

In this section, we present the design of Tracezip. The system architecture is illustrated in Figure 3, in which we add a *compression module* (⑦) to the service side and a corresponding *decompression module* (⑧) to the entry of the backend trace collectors. In the compression module, we maintain two data structures, namely, a Span Retrieval Tree (SRT) and a dictionary, to constantly capture the redundancy across the spans. Upon the generation of a span at the tracepoint, it undergoes compression utilizing the above data structures. If the span carries a new redundancy pattern, it will be seamlessly integrated into the SRT and dictionary. This integration is crucial as it enriches the structures, thereby facilitating the compression for subsequent spans. We accelerate the above process by employing a combination of mapping and hashing techniques. At the decompression module, the spans are restored to their original form by referring to the SRT and the dictionary. To ensure a consistent and reliable data transmission, it is imperative that these data structures are accurately synchronized between the service and backend sides. To achieve this, we develop a differential update mechanism (⑨). This mechanism is designed to precisely pinpoint and propagate only the incremental changes in the data structures, ensuring an efficient synchronization process that minimizes overhead while maximizing data consistency.

3.2 Span Format Conventions

To compress spans by leveraging their recurring patterns, we first stipulate the format of a span. For simplicity and readibility, we assume that a span adheres to the standard JSON data format that defines it as a structured set of key-value pairs. The key is a string, while the value can be either primitive types (strings, numbers, booleans, and null) or two structured types (nested key-value pairs and arrays). This aligns with the format specifications used in many tracing frameworks and tools, e.g., OpenTelemetry [31], Jaeger [19], Zipkin [54]. Typical fields (keys) of a span include: *Name* (a human-readable string representing the operation done), *Parent Span ID* (the span that caused the creation of this span, empty for root spans), *Start and End Timestamps* (the start and end time of the span), *Span Context* (the context of the span including the trace ID, the span ID, etc.), *Attributes* (key-value pairs representing additional information about the span), *Span Events* (structured log messages/annotations on a span), etc. It is important to note that our proposed algorithm is not restricted to JSON or any particular serialization format. For example, TraceZip can work effectively when Protobuf (Protocol Buffers) [2] is used for trace data serialization. With Protobuf's powerful deserialization capabilities, we can leverage its reflection-like APIs or direct-access methods to dynamically access the fields and values of spans. Additionally, Protobuf is designed to be backward and forward compatible, allowing us to modify the message definition by adding or removing fields while maintaining compatibility with older data. Such operations are essential for reducing trace redundancy, e.g., removing span elements that are deemed repetitive.

We also assume that spans possess *structural locality*, meaning that during the continuous execution of a service or component, all spans sharing a common span Name will exhibit an identical structure. In other words, spans with the same Name will consistently retain the same set of keys (e.g., attributes, tags, and metadata), differing only in the specific values associated with them. This assumption arises naturally from the way distributed tracing systems operate, where spans typically represent predefined operations or events within the service workflow. These operations are implemented as part of the service's codebase, which enforces a fixed schema or structure for spans generated by specific instrumentation points. This structural consistency allows for reliable trace analysis, optimization, and redundancy reduction, as the predictability of span structures minimizes the need for per-span schema discovery during processing.

3.3 Span Retrieval Compression and Uncompression

A straightforward approach to compressing spans involves the use of a dictionary. This method creates a dictionary where every unique key and value is assigned a unique identifier. During the compression process, the keys and values of each span are substituted by the corresponding identifiers. The size of the span can then be reduced as the identifiers are much smaller than the original data. However, as revealed by our empirical study, there can still be redundant information among the identifiers. The pure dictionary approach compresses data on a one-to-one basis, i.e., one identifier corresponds to one KV pair. If multiple spans share a collection of common key-value pairs, it is possible to utilize a single identifier to represent this entire set of shared pairs, thereby amplifying the compression efficiency. Thus, we propose to leverage the correlations among the values of spans to further eliminate repetitive information.

Our idea is that for spans generated in each service instance, we organize their key-value pairs as a prefix-tree-like data structure, i.e., SRT. The SRT functions as a multi-way tree, with all non-leaf nodes (except for the root) associated with a key-value pair. For each type of span (which is identified by a unique span Name), there is only one leaf node connected to all the last non-leaf nodes stemming from it. This leaf node holds a collection of keys without values. Figure 4 illustrates an example of SRT, where the gray node and the yellow nodes represent the root and the leaves,

Algorithm 1 Span Retrieval Tree (SRT) Reconstruction and Span Compression

Input: a stream of continuously generated *spans*, a threshold ψ
Output: a constructed SRT $\mathcal{T}$, *compressed spans*
Initialize an empty SRT $\mathcal{T}$
for each *span* in *spans* **do**
    if *span Name* not in $\mathcal{T}$ **then**
        Chain all key-value pairs of *span* and add the path to the root of $\mathcal{T}$
        Assign an identifier to this new path
    else
        for each *key* at every depth of $\mathcal{T}$ **do** ▷ traverse $\mathcal{T}$ from the root to the leaf
            Get the corresponding {*key*: *value*} from *span*
            if {*key*: *value*} **does not exist** at the current depth of $\mathcal{T}$ **then**
                Chain the remaining key-value pairs of *span* and extend a new branch from the direct parent node of *key*
                Calculate the number of unique nodes at each depth of $\mathcal{T}$
                Move the keys to the leaf whose unique value number exceeds ψ, i.e., *local fields*
                Reorder the keys of $\mathcal{T}$ based on the ascending number of their unique values
                Reassign path identifiers
                break
            end if
        end for
    end if
    Compress *span* based on the corresponding path identifier and the values of local fields
end for

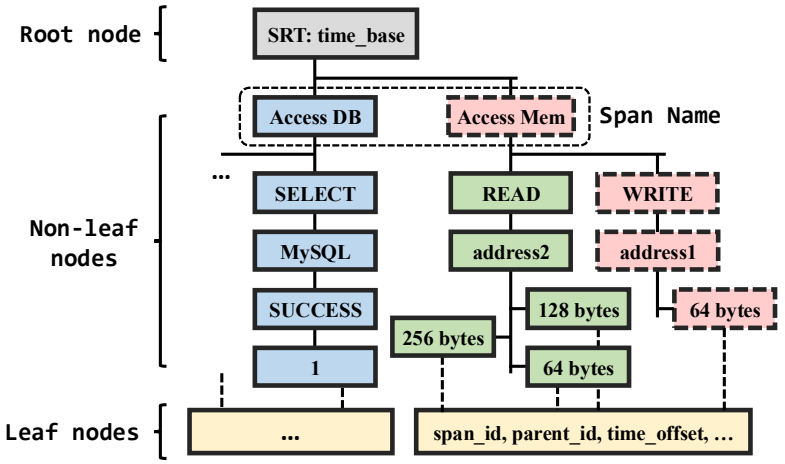

Fig. 4. An Example of Span Retrieval Tree

respectively, while the remaining are the non-leaf nodes. Each span can be "spelled out" by tracing a path from the root down to the leaf. The path of SRT represents the set of KV pairs shared across multiple spans. The non-leaf nodes contain the fields that are more repetitive, which we refer to as *universal fields*. Although spans may exhibit commonality, they will still have some unique KV

Table 1. Span Examples of a Data-processing Service

name	operation	address	data_size	span_id	others
Access Mem	WRITE	address1	64 bytes	id1	...
Access Mem	READ	address2	128 bytes	id2	...
Access Mem	READ	address2	64 bytes	id3	...
Access Mem	WRITE	address1	64 bytes	id4	...
Access Mem	READ	address2	256 bytes	id5	...

(a) Span examples of "Access Mem"

name	type	DB system	status	row.num	others
Access DB	INSERT	MySQL	SUCCESS	1	...
Access DB	SELECT	MySQL	SUCCESS	1	...
Access DB	DELETE	MySQL	SUCCESS	1	...

(b) Span examples of "Access DB"

pairs, such as those related to ID and timestamps. We refer to such pairs as *local fields* and only store their keys at the leaf. The rationale is that such unique fields are incompressible, i.e., not shared with other spans, so we discard their values. *Based on SRT, a span can be represented as a unique path identifier plus its exclusive values that are extracted according to the keys in the leaf node.* Each path identifier collectively represents the KV pairs shared among spans, instead of one identifier for each key and value. Since these common KV pairs constitute a significant portion, the trace size can be substantially reduced, enhancing the overall efficiency.

We present our algorithm for SRT construction and span compression (i.e., Algorithm 1) and explain it using span examples in Table 1. Suppose these spans are continuously generated by different tracepoints of a data-accessing service, including memory and database. Each tracepoint produces a specific type of span with varying attributes. The algorithm takes the stream of spans as input, and the resulting SRT is shown in Figure 4. For each new type of span with a previously unseen span Name, we simply chain all fields of the span (line 6) and add the resulting path to the SRT root. For example, the first row of Table 1-(a) will be structured as $Access\ Mem \hookrightarrow WRITE \hookrightarrow address1 \hookrightarrow 64\ bytes \hookrightarrow id1$ (we omit the keys of the nodes and the other attributes), shown as the pink dashed rectangles. For spans with a known Name, we traverse the SRT from the root to the leaf, and use the key at each depth to retrieve the corresponding key-value pair from the span (line 10). If a retrieved pair does not exist in the SRT, the remaining key-value pairs are chained to construct a sub-path, which is then added as a new branch to the direct parent node (line 12). For example, the second row adds a new path, $READ \hookrightarrow address2 \hookrightarrow 128\ bytes \hookrightarrow id2$, to node *Access Mem*. Each path of SRT will be assigned a unique identifier, as described in Section 4.1.

In SRT, once a new path emerges, we calculate the number of distinct nodes at the same depth (line 13), which represents the number of different values of a key, e.g., the key *span.id* has five distinct values $id1 \backsim id5$. We set a threshold ψ for the size of values a key can have. A key with too many values will be regarded as a local field and moved to the leaf (line 14). For example, *span.id*

will be in the leaf if $\psi = 3$. With the constructed SRT, the fourth row can be compactly represented as the identifier of the first path, i.e., the pink path, coupled with its unique value, i.e., *id4* (line 21). Time-related fields such as span start/end time is also a typical local field. Since spans generated in a short time period will have close temporal fields, we set a *time_base* at the root node, which allows the leaves to store only the offset relative to the time base. This is a common way to compress temporal data. Another special local field is the nested JSON object, such as *{"attributes": {"ip": "172.17.0.1", "port": 26040}}*. We represent the nested structure by prefixing the keys of the child JSON object with the parent's key (e.g., *"attributes-ip": "172.17.0.1"* and *"attributes-port": 26040*), which allows the backend to easily restore the original hierarchy. Technically, spans can extend to any depth as required by the tracing needs. For the consideration of SRT's size, we set a depth limit, which defaults to two, and convert the values of the overly deep keys into pure string type. Similar to other string fields, they will be moved to the leaves if exhibiting too much diversity.

After compression, the span data that needs to be transmitted to the backend trace collector become significantly smaller in size, i.e., only the path identifier and the values of local fields specified by the leaf. At the backend side, the uncompression process to restore the original span is straightforward and efficient. This involves reconstructing the local fields based on the corresponding values received and combining them with the universal fields based on the path identifier. In this process, the backend side should keep the latest copy of the SRT and the value of *time_base*. We introduce an efficient synchronization mechanism later in Section 4.2. For *time_base*, we periodically reset it, e.g., every second, ensuring that the time offset remains consistently small.

3.4 Optimizations for Span Retrieval Tree

So far, we have introduced the algorithms for span compression and uncompression. It can be seen that Tracezip has a small computational complexity. This is because for each span, these processes involve only a single path traversal of the SRT from the root to a leaf. However, the issue of space complexity presents a more significant challenge. The SRT can potentially grow too large and consume an excessive amount of memory. Besides setting a hard constraint on the memory, we have also identified some opportunities to optimize its size.

3.4.1 Span Retrieval Tree Restructuring. During the construction of SRT in Figure 4, we simply follow the left-to-right order of keys in Table 1 to form the parent-child relations among nodes. For example, key *address* is the child of *name* and also the parent of *data.size*. We observe that this may result in a sub-optimal SRT structure. Specifically, for the SRT in Figure 5 which is built based on the spans in Table 1-(b), we can see that the three paths differ only in the *type* field. A better structure can be obtained by moving *type* down to the bottom, which avoids the recurrence of the other three fields. Based on this finding, we propose the following way to restructure the SRT. In Section 3.3, we have calculated the number of possible values associated with each key once a new path emerges. If a parent field has more values than its child, we swap their positions in the SRT. That is, we reorder the keys of SRT based on the ascending number of their unique values (line 15). After reordering, the identical nodes at the same depth will be merged, e.g., *MySQL*, *SUCCESS*, and *1* in Figure 5. Finally, the path identifiers of the restructured SRT will be adjusted (line 16).

3.4.2 Mapping-based Tree Compression. Although we have restructured the SRT to eliminate redundant nodes, there could still be repeated keys and values in it. For example, in Figure 4, key *data.size* appears in all *data.size* nodes, e.g., *{"data.size": "64 bytes"}* and two of them also share value *64 bytes*. Thus, to further compress the size of SRT, we employ a dictionary to map keys/values that occur multiple times to shorter identifiers. We construct the identifiers using the standard alphanumeric set, i.e., [0-9a-zA-Z]. Initially, the hashed output consists of a single character, from '0' to '9,' followed by 'a' through 'z,' and finally 'A' through 'Z.' Upon exhausting the single character

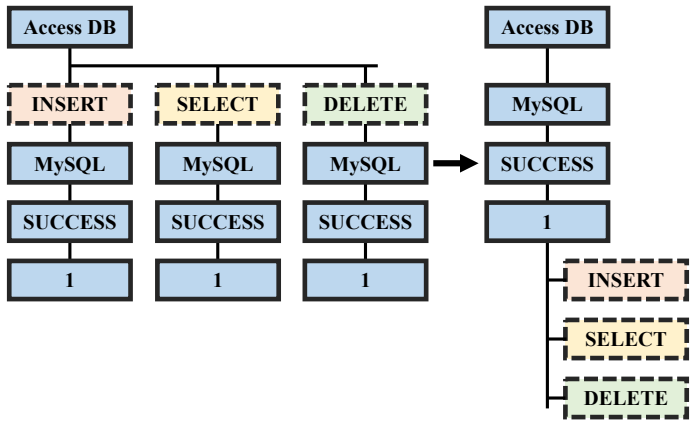

Fig. 5. Span Retrieval Tree Restructuring

possibilities, the function increases the length of the hash output to two characters, starting from '00,' '01,' and so forth. Since each universal field has limited distinct values, i.e., smaller than ψ, the dictionary will also be small in size. Note we do not encode the values of local fields (not in the SRT), which may inevitably make the dictionary too big given their diversity. Similar to the SRT synchronization process between services and the tracing backend, the dictionary will be sent to the backend every time it undergoes an update.

Based on our empirical study (Section 2.2), there exists structural redundancy among the attributes of a span. We address this issue by examining the ingredients of the attributes. Specifically, when constructing the dictionary, we encode the common sub-fields shared among spans (instead of the entire fields) as identifiers. These sub-field identifiers are then used to compose the complete attributes. Take a span example from OpenTelemetry [31], which contains the following fields:

```
{
    "net.transport": "IP.TCP",
    "net.peer.ip": "172.17.0.1",
    "net.peer.port": "51820",
    "net.host.ip": "10.177.2.152",
    "net.host.port": "26040",
    ...
}
```

We can see that in the keys, there are some words that appear multiple times, e.g., `net`, `host`, `port`. Without considering such correlations, we could potentially introduce too much lengthy keys to the dictionary. To remove such redundancy, we first separate each key into a list of tokens based on delimiters dot (".") and underline ("_"), which are configurable. When encoding the key, each of its tokens will be mapped to the corresponding identifier. As the value part exhibits more diversity, we only apply this technique to the keys to avoid too much computational overhead.

4 Implementation

We have implemented TRACEZIP inside the OpenTelemetry Collector with around 3K lines of Golang code. The OpenTelemetry Collector offers a vendor-agnostic implementation of how to

manage telemetry data, which mainly includes four types of components: *exporters*, *processors*, *receivers*, and *extensions*. We implement the span retrieval compression and decompression on the exporter and receiver, which run at the service side and backend side, respectively. The exporter is responsible for building and updating the SRT and dictionary, compressing spans on the fly, and sending them to the remote backend. After accepting the compressed data, the receiver performs span uncompression. We outline some important details concerning the implementation.

4.1 Search Acceleration by Hashing

A straightforward data structure to implement SRT would be linked representation, which enjoys the benefits of dynamic size and efficient alterations (e.g., insertion and deletion). However, in linked representation, the tree nodes are not stored contiguously or nearby in memory, potentially leading to more cache misses. This factor can significantly impede the speed of path search within SRT. To accelerate the search process, we apply hashing to convert each unique path of SRT to a path identifier, which is similar to that in Section 3.4.2. Specifically, for each path, starting from the root we join the values of non-leaf nodes sequentially with a comma separator (similar to the CSV format). Based on the composed path string, we maintain a {*path*: *identifier*} mapping at the exporter. When a new span is generated at the exporter, we extract the values of its universal fields based on the order in SRT. The path search can then be quickly done for the span by checking if its path string exists in the map. We use the `map` data type in Golang, which provides a highly efficient way to achieve this. For any updates to the SRT, we only need to renew the affected paths as discussed in the next subsection.

4.2 Differential Data Synchronization

To ensure reliable span compression and uncompression, the exporter and receiver must maintain consistent copies of both the SRT and dictionary structures. One simple strategy is for the exporter to send the latest versions of these structures upon any update. However, given that updates often affect only a small segment of the overall structures, sending redundant (i.e., unchanged) data with each update would incur network overhead and potentially defer the uncompression process. Thus, we implement a differential update mechanism for more resource-efficient synchronization. The core idea is that at the receiver, instead of maintaining another SRT, we keep a path hashing in the opposite direction, i.e., {*identifier*: *path*}. For any updates to the non-leaf nodes, we can easily pinpoint the affected paths and perform the renewal. For example, in Figure 3, the emergence of a new value (denoted by the pink dashed rectangle) gives rise to a novel path, i.e., *path 3*. In this case, we can add a new entry to the {*path*: *identifier*} mapping at the exporter and sync it with the receiver. For path deletion, the exporter can simply send the corresponding identifier to the receiver for record elimination. Other updates are essentially a combination of path addition and deletion.

For local fields and the mapping dictionary, it suffices to communicate only the changes to the receiver. To ensure that the structures at the receiver is not outdated during the transmission of spans, we leverage the batch processor of OpenTelemetry Collector. It caches the spans sent by SDK until the batch memory is full or its timer expires, instead of immediately forwarding them. After compressing the spans in the buffer, we will make sure that the SRT and dictionary with updates (if any) have been synced with the receiver side before releasing the data.

5 Experimental Evaluation

In this section, we present the evaluation of Tracezip. We first introduce the experimental settings, including the deployed cloud services, the metric for evaluation, and the baseline methods. Next,

we demonstrate the experimental results, which include the effectiveness of trace compression and the analysis of both efficiency and overhead.

5.1 Experimental Setup

5.1.1 Deployed Cloud Systems. To evaluate the compression performance in a realistic environment, we deploy popular cloud systems and collect their traces using the OpenTelemetry Collector instrumented with TraceZip. We serialize the trace data into JSON format and transmit them utilizing HTTP protocols. The selected services include one microservices benchmark named Train Ticket [52] and six open-source application components, including gRPC, Apache Kafka, Servlet, MySQL, Redis, MongoDB.

Train Ticket is a railway ticketing application comprising 41 microservices, each responsible for a specific function, such as user authentication, ticket booking, payment processing, and notification. This benchmark is implemented in different programming languages such as Java, Go, Node.js, Python, etc. Train Ticket allows a comprehensive evaluation in a multifunctional scenario, which has been widely used in many trace-related topics, including trace sampling [4], root cause localization [51, 52], service architecture measurement [35], etc. In order to replicate a live production environment, we employ Locust [27], an open-loop asynchronous workload generator, to drive the services. The workloads are directly borrowed from the original work [52] that introduces the Train Ticket microservices.

The selected six application components have widespread adoption and play critical roles in modern cloud service architectures. They represent a diverse cross-section of the technology stack, which play a foundational role in constructing robust, scalable, and high-performance distributed systems. We generate workloads that reflect real-world usage patterns common in cloud-native and microservices environments. For communication protocols like gRPC and web service frameworks such as Apache HTTP, we simulate typical traffic and user interactions. In messaging systems like Kafka, workloads involve data streaming and message processing, while for data storage solutions like MySQL, Redis, and MongoDB, we focus on common database operations such as read/write transactions. This approach ensures our findings are applicable and relevant to a wide range of real-world scenarios.

5.1.2 Evaluation Metric. To measure the effectiveness of TraceZip, we employ *Compression Ratio* (CR) as the metric, which is widely used in the evaluation of existing compression methods for telemetry data [24, 26]. The definition is given below:

$$CR = \frac{\text{Original File Size}}{\text{Compressed File Size}}$$

In each experiment, we run the same set of workloads, so the size of the original file remains constant. With different compression approaches and configurations, the resulting compressed file may vary in size. As the file size decreases, a higher level of compression is attained, indicating more effective compression performance.

5.1.3 Baseline Methods. Since we are the first to study the problem of trace compression in a live production scenario, there has not been any baseline methods/systems for comparison. Note that TraceZip is orthogonal to existing trace sampling techniques, which compress traces via reducing the volume of data collected. Thus, they cannot be directly compared to TraceZip. In this case, we opt for general-purpose compression algorithms which can be used as out-of-the-box tools to compress traces. Three prevalent and effective algorithms are selected, that is, gzip, bzip2, and lzma. However, as they are not tailored for trace data, suggesting potential for further performance

Table 2. Performance of Trace Compression on Open-source Cloud Systems

	Train Ticket		gRPC		Kafka		Servlet		MySQL		Redis		MongoDB	
	Size	**CR**	**Size**	**CR**	**Size**	**CR**	**Size**	**CR**	**Size**	**CR**	**Size**	**CR**	**Size**	**CR**
Raw	21.0	1	3.08	1	2.47	1	9.36	1	1.88	1	2.01	1	1.10	1
Tracezip	5.19	4.05	0.58	5.31	0.627	3.94	1.45	6.46	0.30	6.27	0.33	6.09	0.21	5.24
gzip	1.93	10.90	0.163	18.91	0.143	17.27	0.506	18.50	0.112	16.85	0.084	23.93	0.065	16.92
Tracezip (gzip)	**1.29**	16.26	0.140	22.81	0.133	18.57	0.396	23.64	0.091	20.61	0.066	30.45	0.051	21.57
improvement	33.0%	1.49x	17.0%	1.17x	7.0%	1.08x	21.7%	1.28x	18.2%	1.22x	21.4%	1.27x	21.5%	1.27x
bzip2	2.41	8.71	0.135	21.96	0.124	19.92	0.421	22.23	0.097	19.38	0.056	35.89	0.054	20.37
Tracezip (bzip2)	1.34	15.62	**0.128**	24.00	0.116	21.29	0.365	25.64	0.087	21.56	0.044	45.68	0.048	22.92
improvement	30.3%	1.75x	8.5%	1.10x	6.5%	1.07x	13.3%	1.15x	10.3%	1.12x	21.4%	1.27x	11.1%	1.13x
lzma	1.93	10.89	0.174	17.67	0.128	19.29	0.487	19.22	0.121	15.61	0.064	31.41	0.065	16.92
Tracezip (lzma)	1.55	13.54	0.146	21.14	0.012	20.58	0.412	22.72	0.097	19.48	0.054	37.2	0.055	20
improvement	35.7%	1.24x	16.4%	1.20x	9.6%	1.07x	20.1%	1.18x	19.9%	1.2x	17.4%	1.19x	22.7%	1.18x

improvement. Our goal is to illustrate the additional compression benefits that Tracezip can provide when applied in conjunction with these standard algorithms.

5.2 Effectiveness of Trace Compression

5.2.1 Open-source Cloud Systems. Table 2 presents the compression performance when collecting traces of the microservices benchmark and cloud applications components. For each system, we calculate the total size of traces collected, the size after compression, and the resultant compression ratios (CRs) when applying different compression algorithms. We can see that Tracezip, as a standalone solution, can achieve CRs ranging from 3.94 to 6.46. This demonstrates that Tracezip can remove more amount of redundant information than that shown by our preliminary study in Section 2.2. Traditional compression tools, i.e., gzip, bzip2, and lzma, reduce the file size with a combination of different techniques such as dictionary-based compression and Huffman coding. Among them, bzip2 generally outperforms the others across most systems, with gzip having the least effectiveness. In the Train Ticket benchmark, the tools demonstrate the least effective compression with a CR of roughly 10, while on the cloud application components, they deliver a better performance, attaining a comparable CR of around 20.

When working in conjunction with the general-purpose compression algorithms, Tracezip can provide additional performance gain. In general, the improvement achieved by Tracezip when combined with bzip2 is less pronounced than when paired with other algorithms. This can be attributed to its already superior compression capability, which may reduce the incremental benefits that Tracezip can offer. In the case of the microservices benchmark, namely Train Ticket, Tracezip achieves a more significant performance improvement of 30%~35%. However, the improvement is less substantial in cloud application components, with Apache Kafka demonstrating an enhancement of less than 10%. As mentioned in Section 2.2, the traces generated by Kafka include the data from its message queues, rendering the attributes more random.

So far we can make an important observation: compared to application backend components, general-purpose compression algorithms are less effective for processing the traces from Train Ticket, where TRACEZIP can offer more substantial improvement. Our careful investigation reveals the following important insight. Based on zero-code instrumentation, the spans collected encapsulate many attributes related to network connectivity (as specified by OpenTelemetry semantic conventions), such as the hostname, IP address, and port of the peer server. For instance, MongoDB captures details of the requests; Kafka producers log information about their consumers. Such information provides a comprehensive view of the request's journey across the distributed system. In production systems, the invocations among different services and components constitute a complex graph, with each node potentially connected to dozens or more instances. Our experimental environment may not be able to accurately replicate the conditions of the production scenarios. Consequently, the connectivity information tends to be relatively static, especially for application backend components that operate at infrastructure and platform layer. In this case, both TRACEZIP and traditional algorithms can properly compress such information, reducing the performance gain that TRACEZIP can offer.

On the other hand, Train Ticket comprises tens of microservices, which can form a invocation graph with moderate complexity. Additional, as a service-oriented application, the traces from Train Ticket contain more information related to business logic. These two factors render the traces produced in Train Ticket more diverse, and the compressible information is more scattered. Traditional compression algorithms are limited to exploiting redundant information within a short sliding window (e.g., 32KB in gzip's Deflate algorithm). On the other hand, TRACEZIP utilizes SRT to continuously capture the redundancy patterns across spans in a *global* manner, which can further reduce the redundancy.

5.2.2 Production Trace Data. We also evaluate TRACEZIP using production trace data collected from Alibaba. Compared to existing microservices benchmarks, this dataset represents the call graphs of a large-scale deployment of over 20,000 microservices in production clusters. The participating microservices can be categorized into two types: stateless services and stateful services. Stateless services operate independently of any stored state data, whereas stateful services, including databases and systems like Memcached, are required to maintain state information. There are three types of communication paradigms between pairs of microservices: inter-process communication, remote invocation, and indirect communication. In addition to this diversity, the trace data also exhibit statistical characteristics typical of industry scale. For example, the size of microservice call graphs follows a heavy-tail distribution; there is a non-negligible fraction of hot-spot microservices; and the microservices can form highly dynamic call dependencies at runtime. This real-world application allows us to examine TRACEZIP's efficacy in handling large-scale, complex data, which is crucial for understanding its potential in practical, production-level scenarios.

Figure 6 illustrates the evaluation results. The raw size of the trace data used in our experiments is 26.15GB. The CRs attained by gzip, bzip2, and lzma are 6.55, 7.30, and 8.52, respectively, reducing the data size to 3.99GB, 3.58GB, and 3.07GB. These CRs are marginally lower than those recorded in the Train Ticket benchmark. This observation aligns with our finding in Section 5.2. That is, relying predominantly on local information, traditional compression algorithms might find it challenging to efficiently compress data characterized by significant diversity and complexity. This is where the strength of TRACEZIP becomes evident. TRACEZIP, with its ability to identify global compression opportunities, enhances the compression performance by 35.1%, 43.6%, and 37.8%, respectively. For example, the combination of TRACEZIP and lzma achieves the optimal CR of 13.69, and compresses the data to a minimal size of 1.91GB. This result underscores the benefits introduced by TRACEZIP, especially when dealing with large-scale and intricate trace data.

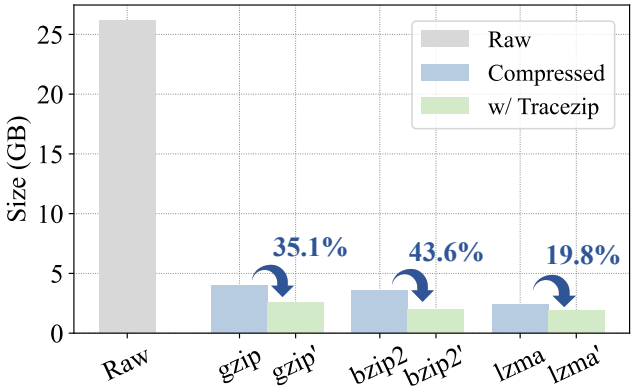

Fig. 6. Compression on Alibaba Production Traces

5.3 Performance Overhead

We examine the overhead of TraceZip from the perspectives of space complexity and computational efficiency.

5.3.1 Space Complexity. At the service side, TraceZip maintains two types of data structures to capture the redundancy among trace spans and perform compression, namely, a SRT along with its hashed paths and a map for dictionary-based compression. To prevent impeding the normal execution of the service, it is imperative that they are constrained in size without excessive memory consumption. To study the space complexity of TraceZip, we select a microservice in Alibaba trace data with diverse spans and calculate the cumulative size of the three data structures after the compression procedure. A critical parameter influencing this size is the threshold ψ, which dictates the maximum number of distinct values for universal attributes. An attribute having an exceeding number of values will be moved to the leaf node, becoming a local attribute. A larger ψ enables TraceZip to compress a broader spectrum of span fields, enhancing performance but at the cost of a more substantial SRT and mapping structure. Conversely, a small ψ compromises the effectiveness but with lower space overhead in return.

Figure 7 illustrates the results, where we can see both the data structure size and compression improvement grow with a larger ψ. In the case of ψ=1,000, SRT and map together take up only 2.56MB of memory, but the performance gain that TraceZip achieves is significant, i.e., 33.8%. This result underscores TraceZip's capacity to achieve substantial compression efficiency while maintaining a balanced memory footprint. However, we notice that the quantities of distinct values that span attributes can have tend to polarize. This can also be observed in Figure 7. The performance plateaus even when ψ=10,000, meaning there is no attributes whose value size falls in the range of [1,000, 10,000]. Certain attributes (e.g., authentication tokens, DB queries, span ID) might possess a substantially larger set of values compared to others. Consequently, their inclusion (when ψ is too large) in the SRT could potentially bloat its size. To address the variability in attribute value distribution and maintain manageable memory usage, we also set a cap on the size of the data structures, e.g., limiting it to 5MB.

5.3.2 Computational Efficiency. To ensure TraceZip can be seamlessly integrated with services, all operations are designed for optimal efficiency. The time complexity of TraceZip's core operations is analyzed as follows. The construction and restructuring of SRT operate with a time complexity of $O(m)$, where m is the number of span attributes. In many scenarios, m typically remains below 20.

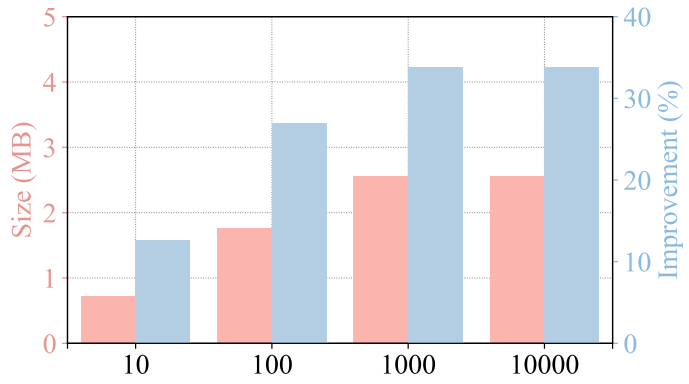

Fig. 7. Performance with Different ψ

Other operations, such as hashing, dictionary mapping, and path searching, all have a complexity of $O(1)$. Thus, the overall time complexity of TRACEZIP is *linear*, making it highly efficient.

To evaluate the efficiency of TRACEZIP, we measure the trace collection throughput for the *basic* microservice [40] of the Train Ticket benchmark. Specifically, we deployed the instrumented OpenTelemetry system within a container (configured with one core and 1GB of memory) to compress and relay the spans. The throughput is calculated as the uncompressed size of spans divided by the time token to transmit the traces from the service to the backend. In the most basic setting, referred to as *Original*, the time is purely the period needed for data transmission and JSON serialization. When compression techniques such as TRACEZIP and gzip are employed, we take into account the additional time required for data compression and decompression. The results are present in Table 3.

Table 3. Performance of Throughput (MB/s) on Train Ticket

ψ	**1**	**10**	**100**	**1,000**
Original	13.98	13.57	13.78	14.05
+TRACEZIP	89.34	94.35	**109.68**	108.15
+gzip	14.65	14.27	14.35	14.02
+TRACEZIP (gzip)	60.65	63.78	68.78	68.56

It can be seen that upon the integration of TRACEZIP, the throughput of trace collection is accelerated by nearly eight times (e.g., from 13.78MB/s to 109.68MB/s). Another interesting observation is that gzip brings little performance gain to the throughput. One important reason is that gzip compression is performed by the HTTP client library before the data is sent over the network. Thus, gzip is applied after the data has been serialized into JSON. As a time-consuming step, JSON serialization constitutes the performance bottleneck. Moreover, gzip indiscriminately attempts to compress all information, including elements such as trace IDs, span IDs, and authentication tokens, which are inherently resistant to compression. The (wasted) computational overhead of gzip compression and decompression thus offsets its benefits. In contrast, TRACEZIP can accurately identify the incompressible attributes, i.e., the local fields, and bypass them. Since TRACEZIP is

applied before JSON serialization, it substantially reduces the volume of data that needs to be encoded. Such a design not only improves the throughput, but also benefits the CPU usage. Our experiments indicate that the CPU utilization of Tracezip is merely 20%∼40% of that in the Original and +gzip settings.

5.4 Threats to Validity

When evaluating the performance and applicability of Tracezip, several potential threats to validity must be considered to ensure the robustness and generalizability of our findings.

Internal validity. One of the primary concerns regarding internal validity is the accuracy of our evaluation metrics and the potential biases in our experimental setup. Real-world cloud services exhibit a vast array of complexities and variations, making it challenging to capture all possible scenarios within a single study. To address this challenge, we carefully select a diverse set of microservices benchmarks and production trace data from Alibaba that we believe are representative of typical cloud service operations. These benchmarks and dataset are chosen to reflect common patterns and behaviors observed in real-world applications, thus providing a meaningful context for evaluating Tracezip's performance. Additionally, any configuration or tuning of Tracezip that is specific to these datasets might inadvertently favor our approach, potentially skewing the results. To mitigate this, we ensure that the benchmarks and dataset are selected and configured independently of Tracezip's development process.

External validity. External validity pertains to the generalizability of our results to other settings or systems. Our evaluation of Tracezip is specifically designed to address the diversity inherent in real-world cloud systems. We implement and test Tracezip within the OpenTelemetry Collector framework and evaluate it across a range of cloud environments and backend components, including gRPC, Apache Kafka, MySQL, and others. These settings are carefully selected to reflect the variety of systems and technologies commonly used in cloud services, ensuring a comprehensive basis for assessing Tracezip's effectiveness. By choosing such a diverse array of environments and applications, we aim to capture the broad spectrum of redundancy patterns and data characteristics found in typical cloud systems. This approach helps to ensure that our findings are applicable to a wide range of real-world scenarios, demonstrating Tracezip's capability to perform effectively in diverse and dynamic cloud environments.

6 Related Work

6.1 Distributed Tracing Systems

Distributed tracing offers a holistic, end-to-end perspective on data requests as they navigate the myriad services within a distributed application. Pioneering works, such as Magpie [1], Whodunit [3], X-Trace [11], and Dapper [39], have laid the foundation of tracing in distributed systems. Recent studies take steps further to address specific challenges to pursue accurate and efficient tracing capabilities. For example, Pivot Tracing [29] and Canopy [20] emphasize cross-application and cross-platform tracing to ensure seamless monitoring and diagnostics across diverse environments. Panorama [16] achieves more sophisticated observability by collecting inter-process and inter-thread error signals. DeepFlow [38] is a non-intrusive distributed tracing framework for troubleshooting microservices. It establishes a network-centric tracing plane with eBPF technique in the kernel. OpenTelemetry [30] standardizes the APIs for telemetry data collection, instrumentation libraries, and semantic conventions. It can be used with a broad variety of observability backends, such as Jaeger [19], Zipkin [54], and Prometheus [36].

6.2 Trace-based System Management

Besides the infrastructure for distributed tracing, trace data are extensively used in many system reliability management tasks. The empirical study presented in [52] demonstrates that the current industrial practices of microservice debugging can be enhanced by integrating appropriate tracing and visualization techniques. MEPFL [53] leverages system trace logs to perform latent error prediction and fault localization for microservice applications. Some works employ traces to construct the dependency graph of microservices for root cause analysis. This include machine learning-based approaches, such as random walk [21], PageRank [45], hierarchical clustering and K-means [47], and spectral analysis [51], etc. In recent years, trace-based fault diagnosis also resorts to deep learning-based approaches, e.g., [12] combines CNN with LSTM to address the complexity of performance debugging. TraceAnomaly [51] uses deep Bayesian network to localize anomalous services in an unsupervised way based on trace representation learning. Some work [8, 23, 46] utilizes a multi-modal approach, which integrates traces with logs [6, 10, 15] and metrics [7, 50] to provide more comprehensive information about system status for microservice troubleshooting. Traces also serve a crucial role in in analyzing system dependencies [28, 44], critical paths [49], resource characterization [25, 41], and microservice architecture [18, 35].

6.3 Trace Sampling and Compression

In production environments, traces can carry comprehensive details, which, if not managed properly, can lead to significant overheads and potentially impact system performance. To reduce overheads, the *de facto* practice is to capture fewer traces. Different from head-based sampling with random trace collection, tail-based solutions enable biased sampling towards more informative and uncommon traces. Some learning-based approaches [4, 17, 22] have been proposed in this field. Sieve [17] employs Robust Random Cut Forest (RRCF) algorithm, a variant of Isolation Forest, to calculate an attention score for each trace, which is then used to determine its sampling probability. Sifter [22] captures edge-case traces by learning an unbiased, low-dimensional model to reconstruct the fixed-length sub-paths of traces. A larger reconstruction loss indicates a higher sampling probability. SampleHST [13] and Perch utilize clustering techniques to divide traces into different groups, and sampling is then performed in each group. STEAM [14] preserves system observability by sampling mutually dissimilar traces. It employs Graph Neural Networks (GNN) for trace representation, and requires human labeling to incorporate domain knowledge. Hindsight [48] introduces the idea of retroactive sampling, which combines the advantages of head-based and tail-based sampling. Specifically, it allows the tracing for all requests at the service side, but reports trace data only after outlier symptoms are detected.

Different from prior research, TraceZip introduces an innovative methodology for mitigating the overhead associated with distributed tracing. Rather than forecasting the significance of individual traces, TraceZip focuses on the compression of spans by leveraging their inherent redundancy. Our approach can work with existing techniques complementarily to enhance the tracing performance. It efficiently manages data volume while ensuring that the essential insights provided by trace data are preserved.

7 Conclusion

In this paper, we propose TraceZip, an online and scalable solution to mitigate the computational and storage overhead associated with distributed tracing. Existing trace sampling techniques either sacrifices the accurate detection of edge cases or tracing scalability. As an orthogonal approach, TraceZip reduces tracing overhead by compressing trace spans into a concise representation, substantially reducing the costs of trace transmission and storage. This is achieved by removing

the redundant data shared across multiple spans. The full trace spans can then be seamlessly restored by retrieving the common data that are already delivered by previous spans. At the service side, such redundancy is continuously encapsulated by Span Retrieval Tree (SRT), which will be synchronized with the backend to ensure consistent trace compression and uncompression. To manage the complexity of SRT, Tracezip constantly restructures SRT into its optimal form and employs mapping technique to further reduces its size. Tracezip also encompasses a differential update mechanism for efficiently synchronize SRT between services and the backend. We have implemented Tracezip within OpenTelemetry Collector and evaluated it on open-source cloud services and production trace data. Experimental results highlight its potential to pave the way for efficient system monitoring.

Acknowledgments

The work described in this paper was supported by the National Natural Science Foundation of China (No. 62402536). We extend our sincere gratitude to the anonymous reviewers for their insightful feedback.

References

[1] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. 2003. Magpie: Online Modelling and Performance-aware Systems. In *Proceedings of HotOS'03: 9th Workshop on Hot Topics in Operating Systems, May 18-21, 2003, Lihue (Kauai), Hawaii, USA*, Michael B. Jones (Ed.). USENIX, 85–90. https://www.usenix.org/conference/hotos-ix/magpie-online-modelling-and-performance-aware-systems

[2] Protocol Buffers. 2024. *Protocol Buffers: language-neutral, platform-neutral extensible mechanisms for serializing structured data.* Retrieved August, 2024 from https://protobuf.dev/

[3] Anupam Chanda, Alan L. Cox, and Willy Zwaenepoel. 2007. Whodunit: transactional profiling for multi-tier applications. In *Proceedings of the 2007 EuroSys Conference, Lisbon, Portugal, March 21-23, 2007*, Paulo Ferreira, Thomas R. Gross, and Luís Veiga (Eds.). ACM, 17–30. doi:10.1145/1272996.1273001

[4] Zhuangbin Chen, Zhihan Jiang, Yuxin Su, Michael R. Lyu, and Zibin Zheng. 2024. Tracemesh: Scalable and Streaming Sampling for Distributed Traces. In *17th IEEE International Conference on Cloud Computing, CLOUD 2024, Shenzhen, China, July 7-13, 2024*, Rong N. Chang, Carl K. Chang, Jingwei Yang, Nimanthi L. Atukorala, Zhi Jin, Michael Sheng, Jing Fan, Kenneth Fletcher, Qiang He, Tevfik Kosar, Santonu Sarkar, Sreekrishnan Venkateswaran, Shangguang Wang, Xuanzhe Liu, Seetharami Seelam, Chandra Narayanaswami, and Ziliang Zong (Eds.). IEEE, 54–65. doi:10.1109/CLOUD62652.2024.00016

[5] Zhuangbin Chen, Yu Kang, Liqun Li, Xu Zhang, Hongyu Zhang, Hui Xu, Yangfan Zhou, Li Yang, Jeffrey Sun, Zhangwei Xu, Yingnong Dang, Feng Gao, Pu Zhao, Bo Qiao, Qingwei Lin, Dongmei Zhang, and Michael R. Lyu. 2020. Towards intelligent incident management: why we need it and how we make it. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1487–1497. doi:10.1145/3368089.3417055

[6] Zhuangbin Chen, Jinyang Liu, Wenwei Gu, Yuxin Su, and Michael R. Lyu. 2021. Experience Report: Deep Learning-based System Log Analysis for Anomaly Detection. *CoRR* abs/2107.05908 (2021). arXiv:2107.05908 https://arxiv.org/abs/2107.05908

[7] Zhuangbin Chen, Jinyang Liu, Yuxin Su, Hongyu Zhang, Xiao Ling, and Michael R. Lyu. 2022. Adaptive Performance Anomaly Detection for Online Service Systems via Pattern Sketching. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 61–72. doi:10.1145/3510003.3510085

[8] Zhuangbin Chen, Jinyang Liu, Yuxin Su, Hongyu Zhang, Xuemin Wen, Xiao Ling, Yongqiang Yang, and Michael R. Lyu. 2021. Graph-based Incident Aggregation for Large-Scale Online Service Systems. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 430–442. doi:10.1109/ASE51524.2021.9678746

[9] Yingnong Dang, Qingwei Lin, and Peng Huang. 2019. AIOps: real-world challenges and research innovations. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 4–5. doi:10.1109/ICSE-COMPANION.2019.00023

[10] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications*

Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 1285–1298. doi:10.1145/3133956.3134015

[11] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. 2007. X-Trace: A Pervasive Network Tracing Framework. In *4th Symposium on Networked Systems Design and Implementation (NSDI 2007), April 11-13, 2007, Cambridge, Massachusetts, USA, Proceedings*, Hari Balakrishnan and Peter Druschel (Eds.). USENIX. http://www.usenix.org/events/nsdi07/tech/fonseca.html

[12] 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 *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck (Eds.). ACM, 19–33. doi:10.1145/3297858.3304004

[13] Alim Ul Gias, Yicheng Gao, Matthew Sheldon, José A. Perusquía, Owen O'Brien, and Giuliano Casale. 2023. SampleHST: Efficient On-the-Fly Selection of Distributed Traces. In *NOMS 2023, IEEE/IFIP Network Operations and Management Symposium, Miami, FL, USA, May 8-12, 2023*. IEEE, 1–9. doi:10.1109/NOMS56928.2023.10154383

[14] Shilin He, Botao Feng, Liqun Li, Xu Zhang, Yu Kang, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. 2023. STEAM: Observability-Preserving Trace Sampling. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 1750–1761. doi:10.1145/3611643.3613881

[15] Shilin He, Pinjia He, Zhuangbin Chen, Tianyi Yang, Yuxin Su, and Michael R. Lyu. 2022. A Survey on Automated Log Analysis for Reliability Engineering. *ACM Comput. Surv.* 54, 6 (2022), 130:1–130:37. doi:10.1145/3460345

[16] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. 2018. Capturing and Enhancing In Situ System Observability for Failure Detection. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 1–16. https://www.usenix.org/conference/osdi18/presentation/huang

[17] Zicheng Huang, Pengfei Chen, Guangba Yu, Hongyang Chen, and Zibin Zheng. 2021. Sieve: Attention-based Sampling of End-to-End Trace Data in Distributed Microservice Systems. In *2021 IEEE International Conference on Web Services, ICWS 2021, Chicago, IL, USA, September 5-10, 2021*, Carl K. Chang, Ernesto Daminai, Jing Fan, Parisa Ghodous, Michael Maximilien, Zhongjie Wang, Robert Ward, and Jia Zhang (Eds.). IEEE, 436–446. doi:10.1109/ICWS53863.2021.00063

[18] Darby Huye, Yuri Shkuro, and Raja R. Sambasivan. 2023. Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows. In *2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*, Julia Lawall and Dan Williams (Eds.). USENIX Association, 419–432. https://www.usenix.org/conference/atc23/presentation/huye

[19] Jaeger. 2024. *An open source, distributed tracing platform.* Retrieved March, 2024 from https://www.jaegertracing.io/

[20] Jonathan Kaldor, Jonathan Mace, Michal 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 *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 34–50. doi:10.1145/3132747.3132749

[21] Myunghwan Kim, Roshan Sumbaly, and Sam Shah. 2013. Root cause detection in a service-oriented architecture. In *ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '13, Pittsburgh, PA, USA, June 17-21, 2013*, Mor Harchol-Balter, John R. Douceur, and Jun Xu (Eds.). ACM, 93–104. doi:10.1145/2465529.2465753

[22] Pedro Henrique B. Las-Casas, Giorgi Papakerashvili, Vaastav Anand, and Jonathan Mace. 2019. Sifter: Scalable Sampling for Distributed Traces, without Feature Engineering. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*. ACM, 312–324. doi:10.1145/3357223.3362736

[23] Cheryl Lee, Tianyi Yang, Zhuangbin Chen, Yuxin Su, and Michael R. Lyu. 2023. Eadro: An End-to-End Troubleshooting Framework for Microservices on Multi-source Data. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1750–1762. doi:10.1109/ICSE48619.2023.00150

[24] Xiaoyun Li, Hongyu Zhang, Van-Hoang Le, and Pengfei Chen. 2024. LogShrink: Effective Log Compression by Leveraging Commonality and Variability of Log Data. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 23:1–23:12. doi:10.1145/3597503.3608129

[25] Jinyang Liu, Zhihan Jiang, Jiazhen Gu, Junjie Huang, Zhuangbin Chen, Cong Feng, Zengyin Yang, Yongqiang Yang, and Michael R. Lyu. 2023. Prism: Revealing Hidden Functional Clusters from Massive Instances in Cloud Systems. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 268–280. doi:10.1109/ASE56229.2023.00077

[26] Jinyang Liu, Jieming Zhu, Shilin He, Pinjia He, Zibin Zheng, and Michael R. Lyu. 2019. Logzip: Extracting Hidden Structures via Iterative Clustering for Log Compression. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 863–873. doi:10.1109/ASE.2019.00085

[27] Locust. 2024. *An open source load testing tool.* Retrieved August, 2024 from https://locust.io/

[28] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, Carlo Curino, Georgia Koutrika, and Ravi Netravali (Eds.). ACM, 412–426. doi:10.1145/3472883.3487003

[29] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot tracing: dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, Ethan L. Miller and Steven Hand (Eds.). ACM, 378–393. doi:10.1145/2815400.2815415

[30] OpenTelemetry. 2024. *High-quality, ubiquitous, and portable telemetry to enable effective observability.* Retrieved August, 2024 from https://opentelemetry.io/

[31] OpenTelemetry. 2024. *OpenTelemetry Traces.* Retrieved July, 2024 from https://opentelemetry.io/docs/concepts/signals/traces/#spans

[32] OpenTelemetry. 2024. *OpenTelemetry Zero-code Instrumentation.* Retrieved August, 2024 from https://opentelemetry.io/docs/zero-code/

[33] OpenTelemetry. 2024. *Trace Semantic Conventions.* Retrieved October, 2024 from https://opentelemetry.io/docs/specs/semconv/general/trace/

[34] OpenTelemetry. 2024. *Vendor-agnostic way to receive, process and export telemetry data.* Retrieved August, 2024 from https://opentelemetry.io/docs/collector/

[35] Xin Peng, Chenxi Zhang, Zhongyuan Zhao, Akasaka Isami, Xiaofeng Guo, and Yunna Cui. 2022. Trace analysis based microservice architecture measurement. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 1589–1599. doi:10.1145/3540250.3558951

[36] Prometheus. 2024. *Monitoring system & time series database.* Retrieved August, 2024 from https://prometheus.io/

[37] Kirk Rodrigues, Yu Luo, and Ding Yuan. 2021. CLP: Efficient and Scalable Search on Compressed Text Logs. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, Angela Demke Brown and Jay R. Lorch (Eds.). USENIX Association, 183–198. https://www.usenix.org/conference/osdi21/presentation/rodrigues

[38] Junxian Shen, Han Zhang, Yang Xiang, Xingang Shi, Xinrui Li, Yunxi Shen, Zijian Zhang, Yongxiang Wu, Xia Yin, Jilong Wang, Mingwei Xu, Yahui Li, Jiping Yin, Jianchang Song, Zhuofeng Li, and Runjie Nie. 2023. Network-Centric Distributed Tracing with DeepFlow: Troubleshooting Your Microservices in Zero Code. In *Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM 2023, New York, NY, USA, 10-14 September 2023*, Henning Schulzrinne, Vishal Misra, Eddie Kohler, and David A. Maltz (Eds.). ACM, 420–437. doi:10.1145/3603269.3604823

[39] 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. (2010).

[40] Train Ticket. 2024. *Train Ticket: A Benchmark Microservice System.* Retrieved August, 2024 from https://github.com/FudanSELab/train-ticket

[41] Kangjin Wang, Ying Li, Cheng Wang, Tong Jia, Kingsum Chow, Yang Wen, Yaoyong Dou, Guoyao Xu, Chuanjia Hou, Jie Yao, and Liping Zhang. 2022. Characterizing Job Microarchitectural Profiles at Scale: Dataset and Analysis. In *Proceedings of the 51st International Conference on Parallel Processing, ICPP 2022, Bordeaux, France, 29 August 2022 - 1 September 2022*. ACM, 47:1–47:11. doi:10.1145/3545008.3545026

[42] Rui Wang, Devin Gibson, Kirk Rodrigues, Yu Luo, Yun Zhang, Kaibo Wang, Yupeng Fu, Ting Chen, and Ding Yuan. 2024. uSlope: High Compression and Fast Search on Semi-Structured Logs. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 529–544. https://www.usenix.org/conference/osdi24/presentation/wang-rui

[43] Junyu Wei, Guangyan Zhang, Yang Wang, Zhiwei Liu, Zhanyang Zhu, Junchao Chen, Tingtao Sun, and Qi Zhou. 2021. On the Feasibility of Parser-based Log Compression in Large-Scale Cloud Systems. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, Marcos K. Aguilera and Gala Yadgar (Eds.). USENIX Association, 249–262. https://www.usenix.org/conference/fast21/presentation/wei

[44] Yang Wu, Ang Chen, and Linh Thi Xuan Phan. 2019. Zeno: Diagnosing Performance Problems with Temporal Provenance. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, Jay R. Lorch and Minlan Yu (Eds.). USENIX Association, 395–420. https://www.usenix.org/conference/nsdi19/presentation/wu

[45] Guangba Yu, Pengfei Chen, Hongyang Chen, Zijie Guan, Zicheng Huang, Linxiao Jing, Tianjun Weng, Xinmeng Sun, and Xiaoyun Li. 2021. MicroRank: End-to-End Latency Issue Localization with Extended Spectrum Analysis in Microservice Environments. In *WWW '21: The Web Conference 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021*, Jure Leskovec, Marko Grobelnik, Marc Najork, Jie Tang, and Leila Zia (Eds.). ACM / IW3C2, 3087–3098. doi:10.1145/3442381.3449905

[46] Guangba Yu, Pengfei Chen, Yufeng Li, Hongyang Chen, Xiaoyun Li, and Zibin Zheng. 2023. Nezha: Interpretable Fine-Grained Root Causes Analysis for Microservices on Multi-modal Observability Data. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 553–565. doi:10.1145/3611643.3616249

[47] Guangba Yu, Zicheng Huang, and Pengfei Chen. 2023. TraceRank: Abnormal service localization with dis-aggregated end-to-end tracing data in cloud native systems. *J. Softw. Evol. Process.* 35, 10 (2023). doi:10.1002/SMR.2413

[48] Lei Zhang, Zhiqiang Xie, Vaastav Anand, Ymir Vigfusson, and Jonathan Mace. 2023. The Benefit of Hindsight: Tracing Edge-Cases in Distributed Systems. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*, Mahesh Balakrishnan and Manya Ghobadi (Eds.). USENIX Association, 321–339. https://www.usenix.org/conference/nsdi23/presentation/zhang-lei

[49] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. 2022. CRISP: Critical Path Analysis of Large-Scale Microservice Architectures. In *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, Jiri Schindler and Noa Zilberman (Eds.). USENIX Association, 655–672. https://www.usenix.org/conference/atc22/presentation/zhang-zhizhou

[50] Zhenyu Zhong, Qiliang Fan, Jiacheng Zhang, Minghua Ma, Shenglin Zhang, Yongqian Sun, Qingwei Lin, Yuzhi Zhang, and Dan Pei. 2023. A Survey of Time Series Anomaly Detection Methods in the AIOps Domain. *CoRR* abs/2308.00393 (2023). doi:10.48550/ARXIV.2308.00393 arXiv:2308.00393

[51] Tong Zhou, Chenxi Zhang, Xin Peng, Zhenghui Yan, Pairui Li, Jianming Liang, Haibing Zheng, Wujie Zheng, and Yuetang Deng. 2023. TraceStream: Anomalous Service Localization based on Trace Stream Clustering with Online Feedback. In *34th IEEE International Symposium on Software Reliability Engineering, ISSRE 2023, Florence, Italy, October 9-12, 2023*. IEEE, 601–611. doi:10.1109/ISSRE59848.2023.00033

[52] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2021. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. *IEEE Trans. Software Eng.* 47, 2 (2021), 243–260. doi:10.1109/TSE.2018.2887384

[53] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. 2019. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 683–694. doi:10.1145/3338906.3338961

[54] Zipkin. 2024. *A distributed tracing system.* Retrieved March, 2024 from https://zipkin.io/