Systemizing and Mitigating Topological Inconsistencies in Alibaba's Microservice Call-graph Datasets

Darby Huye
Tufts University
Medford, MA, USA
darby.huye@tufts.edu

Lan Liu
Tufts University
Medford, MA, USA
lan.liu@tufts.edu

Raja R. Sambasivan
Tufts University
Medford, MA, USA
raja@cs.tufts.edu

ABSTRACT

Alibaba's 2021 and 2022 microservice datasets are the only publicly available sources of request-workflow traces from a large-scale microservice deployment. They have the potential to strongly influence future research as they provide much-needed visibility into industrial microservices' characteristics. We conduct the first systematic analyses of both datasets to help facilitate their use by the community. We find that the 2021 dataset contains numerous inconsistencies preventing accurate reconstruction of full trace topologies. The 2022 dataset also suffers from inconsistencies, but at a much lower rate. Tools that strictly follow Alibaba's specs for constructing traces from these datasets will silently ignore these inconsistencies, misinforming researchers by creating traces of the wrong sizes and shapes. Tools that discard traces with inconsistencies will discard many traces. We present Casper, a construction method that uses redundancies in the datasets to sidestep the inconsistencies. Compared to an approach that discards traces with inconsistencies, Casper accurately reconstructs an additional 25.5% of traces in the 2021 dataset (going from 58.32% to 83.82%) and an additional 12.18% in the 2022 dataset (going from 86.42% to 98.6%).

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; *Reliability*; • **Software and its engineering → Traceability**.

KEYWORDS

Distributed tracing, Cloud Computing, Mitigating data loss

ACM Reference Format:
Darby Huye, Lan Liu, and Raja R. Sambasivan. 2024. Systemizing and Mitigating Topological Inconsistencies in Alibaba's Microservice Call-graph Datasets. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3629526.3645043

1 INTRODUCTION

Today, organizations build distributed applications using a microservice architecture [5, 10]. This architecture—which involves decomposing applications' functionalities into many lightweight services that coordinate over well-defined APIs to process user requests—has many advantages. It facilitates development teams' independence, increases deployment velocity, and enables fine-grained scaling [8, 20]. But, apart from this shared understanding, microservice deployments' concrete characteristics are invisible outside of their respective organizations. This lack of visibility depresses research into microservices. Especially affected are efforts on scheduling, problem mitigation, and debugging, which rely on knowing deployments' scale and emergent properties of how and which services interact to process requests. Published microservices research on these topics [9, 12, 23, 26, 27, 29] are often informed by simple testbeds [4, 10, 30], making their applicability to the large-scale organizations that benefit most from microservices questionable.

Large-scale organizations, such as Alibaba [16], Google [24], and Meta [13] recently published quantitative analyses of their respective microservice architectures. These published studies provide much needed insight into the scale and complexity of the architectures as well as detailed studies of request workflows within them—i.e., how services interact to process requests. Many research efforts are using these analyses to inform their work [3, 7]. Unfortunately, these studies do not provide the raw datasets used for their analyses [24] or only provide summary statistics [13]. This prevents the community from independently verifying results, finding new insights themselves, or using the datasets directly in their work.

To address this concern, Alibaba released two datasets capturing traces of request workflows observed in their microservice architecture [1, 2]. Traces are call graphs, where nodes are services and edges indicate caller/callee relationships between them. Additional annotations on nodes and edges include (but are not limited to) response times, communication protocols used, and service instance ID. The datasets store traces in tabular form with rows corresponding to calls between service. The 2021 dataset covers a 12-hour time-period, totaling 20 million traces. The 2022 dataset covers a 13-day period, totaling over 13 billion traces (estimated). These datasets are treasure troves for research and they are already being used extensively [15, 17, 18, 31]. But, without independent analyses to verify datasets' fidelity, researchers run the risk of basing their work on inaccurate trace data.

This paper provides the needed independent analysis. We analyze the entire 2021 dataset and a 12-hour period of the 2022 dataset. We find that numerous 2021 traces are stored in the dataset in ways inconsistent with Alibaba's specifications. The 2022 traces also exhibit inconsistencies, but at a lower rate. As a result of these inconsistencies, trace graphs built from the datasets will not represent requests' workflows, misleading users. We find that the inconsistencies can be explained by two types of events within the distributed-tracing infrastructure [16] responsible for capturing traces: data-loss and context-propagation errors. The former occurs when log messages (or parts of them) representing individual calls are dropped. The latter occurs when services fail to differentiate

This work is licensed under a Creative Commons Attribution-ShareAlike International 4.0 License.

ICPE '24, May 7–11, 2024, London, United Kingdom

© 2024 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0444-4/24/05
https://doi.org/10.1145/3629526.3645043

different calls made on behalf of a single request.

To mitigate these inconsistencies, we present Casper, a trace construction toolkit that uses hidden redundancies in the datasets to recover from data loss and disambiguate merged calls. We show that Casper creates larger and wider traces than other construction methods—e.g., ones that operate unaware of the inconsistencies or which discard traces with inconsistencies.

We present the following contributions:

(1) We identify and systematize cases where trace data stored in the datasets is inconsistent with the stated specifications. Specifically, 99.48% of traces in the 2021 dataset suffers from these inconsistencies and 85.77% of traces the 2022 dataset suffer from missing calls, contradicting values within rows, and rows that appear more times than expected. We discuss how these inconsistencies affect traces' shapes.
(2) We describe how many of these inconsistencies are explained by: *1)* data-loss events and: *2)* services incorrectly propagating trace context to differentiate inter-service calls. We show how redundancies between `rpcid` values in trace context and caller/callee names within datasets' rows allow many inconsistencies to be circumvented.
(3) We present Casper, a trace construction algorithm that circumvents the inconsistencies[1]. For the 2021 dataset, Casper creates traces that have average sizes, max depths, and max widths that are: 1.14x, 1.22x, and 1.08x larger than a construction method that blindly ignores inconsistencies and 2.71x, 1.32x, and 2.02x larger than a method that discards inconsistent traces. It creates an additional 25.5% of traces with complete connectivity between services for the 2021 dataset (increasing the total to 83.82%) and an additional 12.18% with complete connectivity for the 2022 dataset (increasing the total to 98.6%).

2 ALIBABA MICROSERVICE DATASETS

This section describes the Alibaba datasets' tabular format and the specifications describing traces that are stored within in them (§2.1). We also describe how traces can be constructed from the tabular data using the specifications (§2.1). §3 discusses how the datasets are inconsistent from the specifications and their impact on traces constructed assuming consistency.

We start with a brief description of Alibaba's distributed-tracing infrastructure, which was responsible for capturing the traces. Please see Luo et al. [16] and the datasets READMEs [1, 2] for a more detailed description of the tracing infrastructure and the datasets respectively.

Alibaba's distributed-tracing infrastructure for capturing request-workflow traces: Like most distributed-tracing infrastructures [14, 21, 25], Alibaba's infrastructure works by propagating context with requests' execution. For Alibaba, context includes a per-request unique ID (`traceid`) and a per-call path unique ID (called the `rpcid`). The `traceid` uniquely identifies calls made on behalf of a single request. The `rpcid` uniquely identifies calls and their depth within the trace call graph. It is specified as a series of delimiter `'.'` + IDs. Each service adds a delimiter and adds a unique ID before calling a downstream service. The number of delimiters is equal to the depth of the call. When requests execute log messages, records of them are enriched with context and stored in long-term storage. The trace datasets are comprised of the subset of these logs indicating caller/callee relationships.

[1] Source code and sample data: https://doi.org/10.7910/DVN/SS9SIY

2.1 Tabular format & storage specifications

Format: Figure 1a shows a simplified version of the tabular format, which is largely the same for both datasets. It also shows the graph representation of the trace in Figure 1b. Various columns provide information needed to create the trace topology (nodes and edges) and add annotations. Rows represent log messages or edges of the trace graph—i.e., communication calls between an upstream service (caller) and downstream service (callee). Up to two rows may correspond to a single communication call.

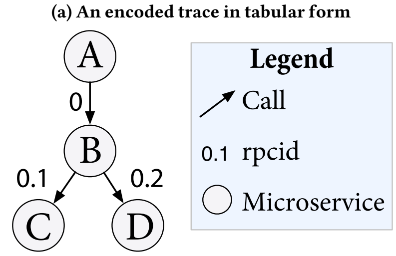
Specifications: The `traceid`, `rpcid`, `UM`, and `DM` columns encode traces' topological information. The first three fields are propagated in context as described above. For a given call the `UM` and `DM` fields identify the corresponding upstream service (caller) and downstream service (callee).

The remaining columns are used to annotate trace nodes or edges. We discuss only ones relevant to our analyses. The `rpctype` column describes the protocol used for a given call. It can be either *RPC*, *HTTP*, *mc* (Memcache), *mq* (Message queue), or *db* (database). The `rt` column describes the call's response time and `ts` denotes a timestamp indicating when the row was recorded by a service. There are two rows for each RPC and HTTP-based call. The first row records the end-to-end response time as measured by the upstream service. The second row records the processing time of the request within the downstream service—i.e., the latency between receiving the request and sending a reply.

Differences between 2021 and 2022 datasets In the 2021 dataset, the row corresponding to end-to-end latency records a positive response time whereas the row corresponding to downstream processing records a negative one [1]. Both response-time values are positive in the 2022 dataset [2]. The 2022 dataset includes additional

	Topology				Annotations	
`ts`	`traceid`	`rpcid`	`UM`	`DM`	`rpctype`	`rt`
166370	1	0	A	B	http	8
166374	1	0	A	B	http	-7
166376	1	0.1	B	C	db	0
166372	1	0.2	B	D	mc	0

(a) An encoded trace in tabular form



(b) Corresponding constructed trace

Figure 1: A trace in tabular form & its constructed version. Not all annotations are shown in the tabular version. Annotations are omitted from the graph. Table follows the 2021 data format.

annotation fields.

2.2 Constructing traces

The construction process described below is general to accommodate traces that start at any arbitrary point of request workflows' execution (i.e., not necessarily at frontend services where workflows typically originate). Responses to questions about the datasets from Alibaba indicate such intermediate starting points are possible [11]. Figure 1b shows the trace that would be constructed from the tabular data in Table 1a.

To build a trace: *1)* Extract all rows of the table with the same `traceid`. *2)* Group rows with the same `rpcid` together as they represent the same call. *2)* Find roots, which are named by the `UM` or `DM` field of calls with the fewest number of `'.'` delimiters in their `rpcid` values. The `UM` is the root if it is defined, else `DM` is the root. The former accommodates intermediate starting points and the latter frontends. *3)* Find calls made by roots, which have the same `rpcid` prefix as roots with one extra `'.'` delimiter, and attach their `DM` values as children. *4)* Repeat step 3 recursively for all leaves in the trace until there are no remaining calls left. Nodes and edges can be optionally annotated during this process.

3 TRACE INCONSISTENCIES

We identified four types of inconsistencies in the Alibaba trace datasets that invalidate the assumptions mentioned in §2. We found these inconsistencies to be prevalent within the datasets with almost all traces having at least one inconsistency. For the remainder of this section, we discuss inconsistencies within the context of a single trace. We focus mainly on the 2021 trace dataset since it has higher error rates, but all inconsistencies discussed were also observed in the 2022 dataset.

Analysis of Alibaba's responses to questions about the datasets [6, 19, 28] indicate that these inconsistencies are explained by data loss and context-propagation errors (CPEs). Data loss results in rows being dropped or fields in rows missing values. Context propagation errors occur when a user does not correctly increment the `rpcid`, assigning the same `rpcid` to many calls. This non-unique `rpcid` is propagated downstream, resulting in downstream calls also having non-unique `rpcid`s. The last subsection (§3.5) gives an example on how many of these inconsistencies may arise given a context propagation error.

Table 1 lists the percentage of unique traces affected by each inconsistency. We next describe each inconsistency in detail.

Inconsistency	2021	2022
Missing duplicate row	99.48%	85.77%
Missing `rpcid`	35.23%	11.42%
Unexpected row	30.16%	6.86%
Contradicting `UM`	33%	7.94%
Contradicting `DM`	26.84%	2.56%
Missing value	94.2%	67.05%

Table 1: Inconsistency frequencies. The portion of traces that have at least once occurrence of each inconsistency.

3.1 Missing rows

There are many missing rows in the trace datasets. We categorize missing rows into two groups: missing duplicate rows and missing `rpcid`s. The duplicate row (i.e. the call or reply) for two-way communication (*http* or *rpc*) is often missing, leaving only one row with either a positive or negative `rt` value. Most traces are missing a duplicate row in both datasets.

A missing `rpcid` is defined to be when we are missing all rows for a `rpcid`. Since `rpcid`s encode topological information about the request workflow, we know all rows for a `rpcid` are missing when we are missing a `rpcid` that is ancestrally between two captured `rpcid`s and is needed to form a call path. We call these missing `rpcid`s internal `rpcid`s (since they are internal nodes in a call graph). A trace may have missing rows before the smallest `rpcid` or after the largest `rpcid`, but there is no way to detect this.

Example: Table 2 gives an example of a missing row and `rpcid`. We are missing the negative `rt` row for the *http* request 0.2. Additionally, we are missing all rows for the `rpcid` 0.2.1, which is the connection between 0.2 and 0.2.1.1.

Implication: Missing duplicate rows does not impact the shape of the trace since the remaining row contains the call path information. We only lose the `rt` for one direction of communication. When we are missing `rpcid`s in a trace, naive rebuilding (using the assumptions outlined in §2) would result in a disconnected trace, under-counting the number of calls and not preserving the true topology.

How to address the inconsistency: Data loss causes us to lose rows, which can present as either a missing duplicate row or a missing `rpcid`. We can use redundant information about the structure of a trace in the `rpcid`s to replace missing internal `rpcid`s, when it's available. For example, in Table 2, the missing `rpcid` 0.2.1 should have `UM` B and `DM` C to form a valid call path.

3.2 Additional unexpected rows

As described in Section 2, `rpcid`s are assumed to be unique for each call in the system. We expect to see one row per message sent; for two-way communication, there should be two rows (call & reply) and for one-way communication there should be one row. Additionally, when we have a reply row for an `rpcid`, all structural information (e.g. `UM`, `DM`, `rpctype`) should be identical since it references a single call. Despite this assumption, we often see additional rows past these thresholds for a single `rpcid`. In the 2021 traces, 42.18% of traces have at least one `rpcid` with unexpected rows.

Example: Table 3 shows an example of additional unexpected rows where 0.3.1 is repeated many times. We have four rows to `DM` C (counting both the + and - `rt` rows) and one row to `DM` D.

`rpcid`	`UM`	`DM`	`rpctype`	`rt`
0.2	A	B	http	+
0.2.1.1	C	D	db	+

Table 2: Missing rpcids. The rpcid 0.2.1 is missing from the table since it's needed to connect 0.2 and 0.2.1.1. Additionally, a duplicate row is missing for the *http* request 0.2.

rpcid	UM	DM	rpctype	rt
0.3	A	B	http	+/-
0.3.1	B	C	rpc	+/-
0.3.1	B	C	rpc	+/-
0.3.1	B	D	mq	+
0.3.1.1	C	E	mq	+

Table 3: Unexpected rows. `rpcid` 0.3.1 is repeated above the expected threshold for the `rpctype`. +/- `rt` is used to indicate we have both the positive and negative rows.

Implication: The assumption that `rpcid`s are unique is invalidated and rebuilding the trace naïvely would under-count the number of unique calls. Additionally, when there are multiple `UM`, `DM` pairs, it's not clear which should be used for the `rpcid`.

How to address the inconsistency Additional unexpected rows are caused by context propagation errors (CPEs), where the user of the tracing infrastructure did not increment the `rpcid` for each unique call. This appears in the table as additional rows with the same `rpcid` and `UM`, but potentially different `DM`s. In Table 3, the CPE originates from service B, which makes at least two calls to service C and one to D. The non-unique `rpcid`s are passed downstream, creating more non-unique `rpcid`s (and call paths) further downstream. For example, 0.3.1.1 (in Table 3) has `UM` C, but we do not know which of B's calls to C made the subsequent call to E.

3.3 Contradicting Values

There are inconsistencies in the datasets where rows that should contain identical values have conflicting values (e.g. the two rows corresponding to the call and reply for a two-way communication call should have the same `UM` and `DM`). We categorize contradicting values into two groups: contradicting `DM`s are when all rows with a `UM` has multiple `DM`s and contradicting `UM`s are when one or more of the `UM`s for an `rpcid` don't match the upstream call's `DM` (i.e. not forming a valid call path). Contradicting `UM`s are independent of the `DM` value. A single `rpcid` can have both types of contradicting values in their rows. A large portion of the 2021 traces (26%) have at least one `rpcid` with contradicting `DM` values and 37% of the traces have contradicting `UM`s.

Example: Table 4 shows two rows for `rpcid` 0.3.1.1. There is conflicting `UM` and `DM` information in the two rows. Since there are multiple `UM` values, at least one of the rows may not connect upstream resulting in an invalid path.

Implication: Naïvely rebuilding trace with contradicting values could create invalid call paths, depending on which row's information is added to the trace topology. Since each unique `rpcid` is assumed to have one `UM` and one `DM`, naïvely rebuilding would not check for this inconsistency.

How to address the inconsistency Contradicting values are the result of context propagation errors. Contradicting `DM`s appear when the user does not increment the `rpcid` when making calls to different downstream services. Upon cursory inspection, we found contradicting `UM`s are either downstream from CPEs or seem to have an incorrect `rpcid` that could be remedied by adding a '.' followed by an integer to connect the call one level downstream. We can use redundant information about the call paths to help determine

rpcid	UM	DM	rpctype	rt
0.3.1.1	C	E	mq	+
0.3.1.1	D	F	mq	+

Table 4: Contradicting values.

accurate `UM`s and `DM`s (or if the `rpcid` is not unique).

3.4 Missing Values

Many values in the datasets are missing or contain `'(?)'/UNKNOWN` as the value. In fact, most traces in both datasets contain at least one missing `UM` or `DM` value (94.2% and 69%).

Implication: Naïvely rebuilding traces with missing values misses opportunities to uncover the true value, resulting in skewed metrics about the frequency of specific microservices.

How to address the inconsistency Missing values are the result of data loss, which is not uncommon in large distributed systems. For two-way communication, there is often a duplicate row for the `rpcid` which contains the missing information. If there is no duplicate row, these missing values can be recovered using call path information from an upstream or downstream call.

3.5 Combination of inconsistencies

Context propagation errors (CPE) often show up as a combination of the inconsistencies. We always see unexpected rows for CPEs, but this is typically combined with contradicting values. For example Table 3 showed both `DM` values C and D, which are contradicting.

Downstream from CPEs, the same `rpcid` is used as a seed for downstream calls. In the Table 3 example, 0.3.1 is the seed `rpcid` for all downstream calls made from C and D. If we added an additional row to this example with `rpcid` 0.3.1.1.1, `UM` X and `DM` Y, we would have a contradicting path (since X is not the same as upstream call 0.3.1.1's `DM` E). To make things more complicated, the contradicting path inconsistency would no longer exist if we assumed we were missing its' upstream `rpcid` (which could have `DM` X). The point here is that CPEs cause many inconsistencies since they invalidate the assumption that `rpcid`s are unique. This makes it challenging (and sometimes impossible) to decipher the trace topology.

4 CASPER

We introduce Casper, which aims to create the largest accurate request workflow topologies. Building on the intuition in §3, we discuss which inconsistencies can be circumvented (e.g., are recoverable), and how to recover from them (§4.1). We also discuss when inconsistencies are not recoverable (§4.2). We then present Casper's algorithm for rebuilding the traces (§4.3). We conclude with limitations of our approach (§4.4). Casper is implemented in 711 lines of Python code. It takes as input all rows for a trace and outputs the constructed trace in Alibaba tabular or OpenTelemetry JSON [21] format.

4.1 Recoverable inconsistencies

4.1.1 Data loss. **Missing internal calls**: Observing missing internal calls due to data loss, we can determine the exact number of missing calls on the call path using the `rpcid` structure. We find the call that is missing its upstream call and recursively drop the trailing '.' from the `rpcid` until we reach an existing `rpcid`. Each new `rpcid` we create by dropping a '.' becomes a call in the trace.

Example: In figure 2, (1) shows how `rpcids` 0.1.1 and 0.1.1.1 are added to the trace to fill the hole between the existing `rpcids`.

Missing values captured in redundant rows: We use information captured in duplicate rows or through call paths to fill in missing `UMs` and `DMs`.

Example: In figure 2, (2) shows a recovered (missing) `rpcid` 0.2.1, which was added to connect the existing `rpcids`. In table format, we can refill 0.2.1's `UM` with `DM` B (from its upstream call) and `DM` C (from its downstream call).

4.1.2 Context propagation errors (unique paths). **Context propagation errors (CPEs) can be fixed at the source**: As stated in §3, a CPE originates from a service which incorrectly uses the same `rpcid` for different calls. We call a CPE's origin the source of the error. By differentiating each `rpcid`, we can always rebuild the trace structure at the source of a CPE.

First, we calculate the number of unique calls that were incorrectly assigned the same `rpcid`. The 2021 traces and 2022 traces use `rts` differently, so we handle each case independently. The 2021 traces use negative `rts` for reply rows. When the `rt` is below a certain threshold, it is rounded down to 0 (both for the call and reply edges). The call `rt` includes the child execution time plus the network latency while the reply `rt` is only the child execution time, so the reply edges must have a `rt` less than or equal to the call `rt`. We use these characteristics to calculate the number of calls for a given `rpctype` to each `DM`. For one-way communication, the `rt` values should all be positive (mostly 0), so the number of calls is equivalent to the number of rows.

For two-way communication in the 2021 dataset, we calculate number of calls as follows:

$$
\begin{aligned}
num\_fast\_calls &= \lfloor \frac{(rt==0)}{2} \rfloor \\
extra\_fast\_row &= (rt==0)\%2 \\
num\_slow\_calls &= \max(-\mathtt{rt}, abs(+\mathtt{rt} - extra\_fast\_row)) \\
num\_calls\_to\_DM &= num\_fast\_calls + num\_slow\_calls
\end{aligned} \tag{1}
$$

`-rt` is the number of rows with negative response times and `+rt` is the number of rows with non-zero positive response times. Since two-way communication should have one positive and one negative row, where the negative row could be 0, we get the minimum number of unique calls if we maximize the number of +/- pairs made. num_fast_calls counts the pairs of rows with 0 `rt`. $extra\_fast\_row$ is 1 if there is an odd number of rows with 0 `rt`. We try to pair any leftover 0 `rt` rows with +`rt` rows. num_slow_calls tries to match the 0 `rt` row with a positive `rt` row (taking the absolute value when there are no positive `rt` rows), and then counts the pairs between the remaining + `rt` rows and - `rt` rows. Taking the max gives us the total number of pairs and the remaining unpaired rows. Finally, we add the fast and slow calls to get the total number.

The 2022 traces only have positive `rts`, the number of calls is calculated by:

$$num\_calls\_to\_DM = ceiling(rt/2) \tag{2}$$

for two-way communication calls (and is the number of rows for one-way communication).

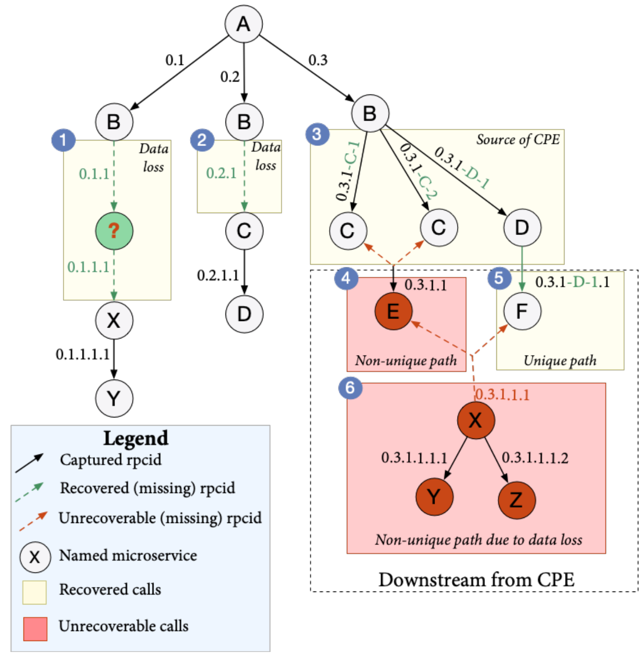


Figure 2: CASPER example trace. Regions of the trace that are corrected are highlighted in yellow and regions that are fundamentally unfixable are highlighted in red.

Since we can always be missing rows, the true number of unique calls is unknown. These calculations determine the minimum number of unique calls.

Using the minimum number of calls to each `DM`, we can fix the topology at the source of a CPE. We do this by updating the `rpcids` to be unique for each call. We modify the `rpcids` to be of the form $rpcid - DM - i$ where i is between 1 and the minimum number of calls to the `DM`.

Example: In figure 2, (3) shows how the `rpcids` are updated to be unique for a CPE. The tabular version of this data (table 3) has five rows for the `rpcid` 0.3.1. We calculate that there are two calls to C and one to D. We update the `rpcids` to be: 0.3.1-C-1, 0.3.1-C-2, and 0.3.1-D-1.

Unique call paths downstream from CPEs: When there is only one call to a `DM` at the source of a CPE, there is a possibility that we can rebuild the trace downstream from this service. If there are multiple calls to a `DM`, we cannot determine which instance of the `DM` made which downstream calls.

All downstream `rpcids` from CPEs are not assumed to be unique because they share a non-unique `rpcid` in their ancestry. As a result, we can only rely on the call depth information from `rpcids` as being accurate. We can use `UM` and `DM` information to rebuild call paths downstream from CPEs when the call path is 1) unique (i.e. the chain of `UM`, `DM`, and call depth information forms a single valid call path) and 2) complete (there is no data loss or unknown values in the call path).

Example: In figure 2, (5) shows a unique path downstream from a CPE that we can connect to the trace. Table 4 shows the tabular version of this portion of the trace, where there is a row for 0.3.1.1 with `UM` D, connecting to the single D node in the trace. We update the `rpcid` to be unique by changing its non-unique ancestor to be unique (e.g. 0.3.1.1 → 0.3.1-D-1.1).

4.2 Unrecoverable inconsistencies

4.2.1 Data loss. **Data loss that has no redundancies**: Most instances of data loss have redundancies in the dataset. However, there are some instances where there are no redundancies and the data is unrecoverable. For example, when sequential `rpcids` are missing, not all `UM` and `DM` information is recoverable.

Example: In figure 2, (1) shows that we are able to recover the missing `rpcids` 0.1.1 and 0.1.1.1, but we cannot determine the service name for the additional node.

4.2.2 Context propagation errors (non-unique paths). **Non-unique call paths downstream from CPEs**: We cannot remedy calls downstream from CPEs when they do not form unique call path. In the presence of data loss, it is fundamentally not possible to replace the missing calls since we cannot determine a unique ancestry to reconnect via. When we have missing `UM` or `DM` values, they are not recoverable since there are often many unique possible values.

Example: In figure 2, (4) shows how the `rpcid` 0.3.1.1 (from table 3) cannot be uniquely connected to the trace. Since we cannot definitively connect this edge to the existing trace, we remove it and all rows downstream from it. Figure 2 (6), which visually represents the data in table 5, is affected by data loss. We are missing a row for the `rpcid` 0.3.1.1.1, which is needed to determine if the node X connects uniquely to the trace via the node F or non-uniquely to the trace via node E.

Conflicting paths, where the `UM` does not connect upstream: As described in section 3.3, conflicting `UMs` have a special case where they are not downstream from a CPE. In this case, there is only one service upstream. Any rows with `UMs` that do not match the upstream's `DM` are dropped as they form invalid call paths.

4.3 Casper algorithm

The Casper algorithm performs a best case reconstruction of the trace topologies and guarantees that the edges in the resulting graphs are accurate. We keep track of the number of unrecoverable `rpcids` that are omitted from the traces. Casper begins with general preprocessing of the data including filling missing values with duplicate rows. The meat of the program is in handling data loss and CPEs, which is outlined below.

At a high-level, Casper performs a breadth first search (BFS) traversal over the `rpcids` in each trace. When it identifies data loss upstream from a `rpcid`, it recursively fills in missing calls until it connects upstream. When it identifies a CPE, it fixes the error at the source and attempts to reconstruct the `rpcids` downstream from the CPE (algorithm 2) before returning to the BFS traversal (algorithm 1).

For each trace, Casper is initialized by sorting the `rpcids` in BFS order and identifying all root `rpcids`, which have no upstream

rpcid	UM	DM	rpctype	rt
0.3.1.1.1.1	X	Y	db	+
0.3.1.1.1.2	X	Z	db	+

Table 5: Unrecoverable call paths downstream from CPE, affected by data loss.

Algorithm 1 Casper algorithm for a single trace

1: $rpcids \leftarrow BFS\ sorted\ rpcids\ for\ this\ trace$
2: $root\_rpcids \leftarrow allroots$
3: **for** `rpcid in rpcids` **do**
4: $upstream\_rpcid = rpcid.rsplit('.', 1)[0]$;
5: **if** Data loss **then**
6: **while** *upstream_rpcid* not in trace **do** '
7: Add edge for *upstream_rpcid*;
8: *upstream_rpcid* ← next *upstream_rpcid*;
9: **end while**
10: **end if**
11: **if** Context propagation error **then**
12: *DMs* ← unique DMs for rpcid;
13: **for** DM in DMs **do**
14: $min\_calls\_to\_DM \leftarrow max(R+, R-)$;
15: **for** i in min_calls_to_DM **do**
16: Add edge for *rpcid_DM_i*;
17: **end for**
18: **end for**
19: Rebuild downstream CP rpcids (Alg 2);
20: **end if** Add edge for *rpcid*;
21: **end for**

`rpcids` (alg 1, lines 1–2). It then iterates over the `rpcids` in BFS order from each root and performs:

(1) Check for data loss: if the `rpcid` is not a root, drop the trailing '`.`' and check for a *upstream_rpcid* (alg 1, lines 4–5).
  (a) If the *upstream_rpcid* does not exist, recursively add calls (with `UM`/`DM` values when possible) until we connect to the trace (alg 1, lines 6–9).
(2) Check for CPE: Calculate the minimum number of calls made by this `rpcid`. If there are more than the expected number of rows (alg 1, line 11):
  (a) Extract the list of unique `DMs` called by the `UM`. For each `DM`, calculate the number of calls to the `DM`. Create a unique `rpcid` for each call to the `DM` of the form `rpcid`=`rpcid`−`DM`−*i* where i ranges from 1 to the number of call (alg 1, lines 12–18).
  (b) Extract all `rpcids` downstream from the CPE and rebuild independently. These rows have different assumptions (i.e. that the `rpcid` is not unique) so must be handled separately (alg 1, line 19).
(3) Add edge to the trace, if no errors (alg 1, line 20)

Algorithm 2 describes how we remedy `rpcids` downstream from CPEs. At a high-level, Casper first identifies and removes subtrees in the trace that are downstream from data loss (i.e. disconnected subtrees). Next, Casper performs call path validation, filtering out non-unique call paths. Algorithm 2 is initialized with only the `rpcids` downstream from the source of a CPE, which are sorted in BFS order (alg 2, lines 1–2).

(1) Check for downstream data loss: For each `rpcid`, check if it has a dangling tree below it. If the `rpcid` has no direct children, but has descendants (alg 2, lines 3–4):
  (a) Get the list of descendant `rpcids` and delete all rows with these `rpcids` (alg 2, line 5–6).

(2) For each *row* that is downstream from the CPE and not affected by data loss, calculate the number of calls this row connects to upstream. This is done by generating the *parent*_`rpcid`, filtering the parent rows that connect to this row's `UM`, and calculating the number of calls for those rows (alg 2, line 9–10):
  (a) If this row connects to a unique call path: update it's `rpcid` to be unique, by replacing its ancestry `rpcid` with its corrected *parent*_`rpcid` (alg 2, line 11–12).
  (b) If this row connects to a multiple call paths: delete the row and any connecting downstream call paths. (alg 2, line 13–14)

Algorithm 2 supports fixing sequential CPEs as long as the call paths are unique.

Algorithm 2 Casper rebuild calls downstream from CPE

1: *rows* ← rows with rpcids downstream from CPE;
2: *rpcids* ←BFS sorted rpcids downstream from CPE;
3: **for** `rpcid in rpcids` **do**
4: **if** No child rpcid exists but exists descendents **then**
5: *decendent_rpcids* ← *rpcid*.∗ from rpcids;
6: delete rows for *decendent_rpcids*;
7: **end if**
8: **end for**
9: **for** `row in rows` **do**
10: *num_connecting_calls* ← num calls row connects upstream to;
11: **if** *num_connecting_calls* == 1 **then**
12: update `rpcid` in row;
13: **else**
14: delete row & connecting downstream rows;
15: **end if**
16: **end for**

4.4 Limitations

Missing `rpcids` before or after all recorded `rpcids`: Missing `rpcids` that are smaller than the smallest `rpcid` in the table or larger than the largest `rpcid` in the table. As mentioned in [6], root `rpcids` can be anything (although it's most commonly 0 or 0.1). Additionally, we found there can be multiple roots in a single trace. When the root `rpcid` is larger than 0.1 (i.e. starting at a 'depth' of greater than 2), it could be the true root of the trace or it could be missing `rpcids` before it.

Corrupted values: We must make assumptions about the trace data that allow us to rebuild the topology. In addition to the assumptions provided by Alibaba, we assume `traceids` are accurate. If `traceids` (or any other fields for that matter), are corrupted in unpredictable ways, we cannot guarantee we will identify it and can remedy it.

5 EVALUATION OF CASPER

We seek to answer the following regarding Casper's efficacy in circumventing inconsistencies in the Alibaba datasets.

Q1: How are traces generated by Casper different from traces generated with other approaches? More specifically, what trace topological characteristics change using different reconstruction approaches?

Q2 How much do the recovery mechanisms in Casper impact trace topologies?

Q3 How many additional complete traces does Casper reconstruct compared to filtering out all traces with any inconsistencies?

The answers to *Q1* are a cautionary tale to researchers using the Alibaba trace dataset that reconstructing traces ignoring errors will lead to vastly skewed results that may impact the design or evaluation of their research artifacts. The answers to *Q2* evaluate the effectiveness of the recovery mechanisms in Casper. The answer to *Q3* informs us how much the built-in side-channel redundancies in the Alibaba traces are able to help correct the inconsistencies without dropping communication calls or filtering entire traces.

For all of these analyses, we use a randomly sampled 10.8% (2,240,550) of the 2021 trace data and 1% (5,079,746) of the 2022 trace data. We use a lower sampling rate for the 2022 dataset because it contains many more traces than the 2021 version. We used six `r6.xlarge` instances to split the datasets as per `traceid` and one `c5a.23xlarge` EC2 instance w/90 threads to run the Casper algorithm and other construction approaches described in this section. The other construction methods are variants on the Casper implementation and run inline with its execution.

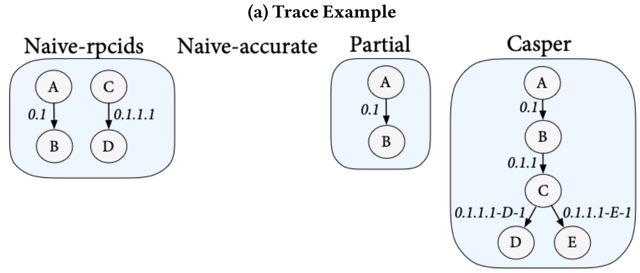
5.1 Comparing construction methods

5.1.1 Methodology. We consider three alternative approaches to Casper for constructing traces without deep knowledge of the inconsistencies in the datasets: `naive-rpcid`, `naive-accurate`, and `partial`. Figure 3 illustrates how the four approaches differ when reconstructing one example trace that has inconsistencies. We describe the alternative rebuild modes below.

`naive-rpcid`: keeps the first occurrence of a unique `rpcid` in the table and neither detects nor recovers any inconsistencies (similar to construction process in §2). Figure 3 shows the first row for each `rpcid` represented as a graph. The trace is disconnected since `rpcid`

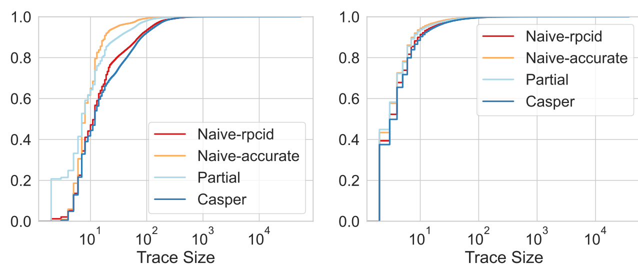
`rpcid`	`UM`	`DM`	`rpctype`	`rt`
0.1	A	B	rpc	+/-
0.1.1.1	C	D	rpc	+/-
0.1.1.1	C	E	db	+

(a) Trace Example



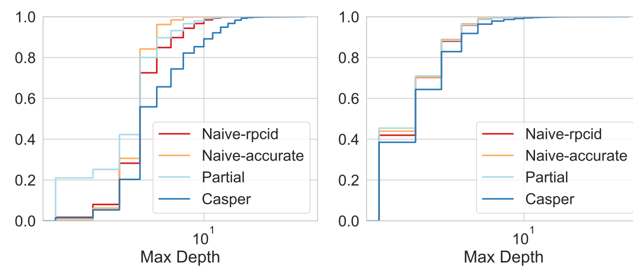
(b) Modes

Figure 3: Trace building modes. Four different modes for building trace graphs, each has a different method of handling errors and inconsistencies.



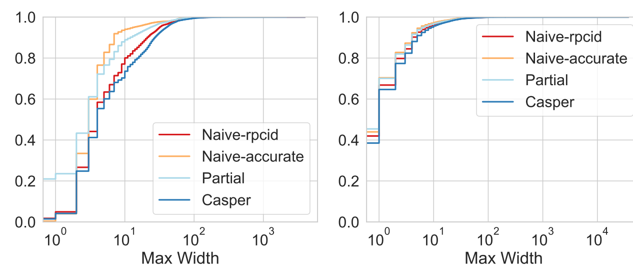
(a) 2021 (b) 2022

Figure 4: Trace sizes for different modes



(a) 2021 (b) 2022

Figure 5: Maximum trace depth for different modes



(a) 2021 (b) 2022

Figure 6: Maximum trace width for different modes

0.1.1 is missing. Additionally, we are missing node E since it was not captured in the first row for its `rpcid`.

`naive-accurate`: detects inconsistencies and ignores the entire trace if an inconsistency is identified. Figure 3 shows once the missing `rpcid` 0.1.1 is detected, all rows for the trace are deleted.

`partial`: keeps calls in the traces that are not affected by an inconsistency. When an inconsistency is identified, the entire downstream call path is removed from the trace. `partial` traces preserve the accurate portions of traces. Figure 3 shows once the missing `rpcid` 0.1.1 is detected, all downstream rows are deleted.

`Casper`: as described in section 4. Figure 3 shows how Casper fixes the missing call 0.1.1 and the CPE at 0.1.1.1, updating the repeated `rpcid` to be unique for each call.

For `naive-accurate` and `partial`, we allow inconsistencies that are trivial to fix (e.g. missing values and missing duplicate rows) since they do not affect the trace topology.

To compare the four approaches, we analyze the following topological characteristics of the reconstructed traces: trace size, call depth, and width. Trace size represents the total number of microservices in the trace. A microservice can be called many times within a trace and each call is included in the trace size. Call depth is the maximum depth of the call paths in the traces. Width is the maximum number of calls made by a single microservices. In graph form, this is the largest fan-out. We compare the cumulative distribution functions (CDFs) for all metrics.

5.1.2 Results. Overall, we find that Casper traces are larger, wider, and deeper than all other methods of constructing traces for both the 2021 and 2022 datasets. The 2022 traces are smaller (size, depth, and width) than the 2021 traces. The 2022 traces have less inconsistencies, so Casper's impact on the trace topology compared to `naive-rpcid` is less significant.

Trace size: For both the 2021 and 2022, Casper builds larger traces than all other approaches by correcting data loss and CPEs. Figure 4 shows the CDF of trace sizes for both years. For 2021, at the 50th percentile (P50), a Casper trace is the same size as `naive-rpcid` traces. However, Casper produces larger traces at P75 than all other modes. On average, Casper traces have size 33.08 whereas the size is 12.22, 15.50, and 29.11 for `naive-accurate`, `partial` and `naive-rpcid` respectively. For 2022, Casper traces have similar, the same or slightly bigger, size compared to all other modes at all percentiles. Traces from 2022 are overall smaller than those from 2021. The average trace size `naive-accurate` is 12.22 in 2021, but is decreased to 4.98 in 2022.

Call depth: For both the 2021 and 2022, Casper builds deeper traces than all other approaches by reconnecting call paths that have missing `rpcids`. Figure 5 shows the CDF of the maximum call depth of a trace for both years. For 2021, at P50, Casper traces have the same depth as the other modes. However, Casper produces deeper traces at P75 than all other modes. On average, Casper traces have depth 6.40 whereas the depth is 4.84, 4.56, and 5.26 for `naive-accurate`, `partial` and `naive-rpcid` respectively. For 2022, Casper traces have the same depth or slightly deeper than all other modes at all percentiles. Traces from 2022 are overall shallower than those from 2021. The average depth for a `naive-accurate` trace is 4.84 in 2021, but is decreased to 3.01 in 2022.

Width: For both the 2021 and 2022, Casper builds wider traces than all other approaches by differentiating repeated `rpcids` generated by CPEs. Figure 6 shows the CDF of maximum width of a trace for both years. For 2021, P25 and P50, a Casper trace has similar width as the other modes. At P75, Casper traces are wider. On average, Casper traces have width 10.73 whereas the width is 5.30, 5.91, and 8.97 for `naive-accurate`, `partial` and `naive-rpcid` respectively. For 2022, Casper traces are similarly wide or slightly wider than all other modes at all percentiles. Traces from 2022 are overall narrower than those from 2021. The average width for `naive-accurate` traces is 5.30 in 2021, but is decreased to 1.92 in 2022.

5.2 Impact of recovery mechanisms

5.2.1 Methodology. We break down Casper's recovery mechanisms (described in §4.3) into four parts and quantify their impact. 1) *Adds missing calls* counts the number of new calls added to a trace. 2) *Fills in missing values* counts the originally unknown microservice names that Casper recovers. 3) *Updates* `rpcids` *at a CPE source* measures the number of `rpcids` added when differentiating calls at the first occurrence of a CPE in a call path. 4) *Recovers* `rpcids` *downstream from a CPE source* counts the number of calls Casper identified as uniquely connected to the trace downstream from the first CPE in the call path.

Casper collects metrics for each of the recovery mechanisms when reconstructing the traces. We calculate the number of traces that are affected by each recovery and its impact within the trace.

5.2.2 Results. **Adds missing calls**: Casper recovers all missing internal calls in a trace. For 2021, 30.47% of traces have at least one missing internal `rpcid`, with the average number of calls 10.7 (std: 23.79, P99: 115). For 2022, 8.77% of traces have at least one missing call added by Casper, with average 3.57 (std: 13.12, P99: 30). Casper reconnects broken traces, resulting in longer call paths.

Fills in missing values: Casper fills in missing `DM` values using duplicate information stored in rows or call paths. For 2021 traces, we are able to recover at least one `DM` in 99.98% of traces. On average, traces have 2.84 recovered `DM` names (std: 5.61, P99: 27). For 2022 traces, we are able to recover at least one `DM` in all traces. On average, traces have 2.61 recovered `DM` names (std: 6.99, P99: 34).

Updates `rpcids` at a CPE source: Given a call path starting from the root call, the first occurrence of a CPE is a CPE source. Casper modifies `rpcids` to be unique to differentiate different calls that originally shared the same `rpcid`. This modification may preserve more branches and yield wider traces. For 2021 traces, Casper modifies on average 2.77 `rpcids` per CPE source (std: 2.65 and P99: 11). For 2022 traces, Casper modifies on average 2.02 `rpcids` at a CPE source (std: 0.18 and P99: 3).

Recovers `rpcids` downstream from a CPE source: Casper connects unique call paths downstream from a CPE source, updating their `rpcids` to be unique which may preserve longer call path and yields deeper traces. We measure this impact by counting the number of such updated `rpcids` per CPE source. For 2021 traces, Casper modified on average 3.08 downstream `rpcids` (std: 11.22 and P99: 48). For 2022 traces, Casper modified on average 1.11 downstream `rpcids` (std: 32.97 and P99: 19). Note that additional CPEs can occur downstream, but they are rare. For 2021, the average number downstream CPEs is 0.23, (std: 1.5 and P99: 6). For 2022, the average number downstream CPEs is 0.14, (std: 2.71 and P99: 3).

5.3 Additional complete traces

5.3.1 Methodology. We evaluate Casper's effectiveness at rebuilding complete traces by measuring the number of additional complete traces output by Casper (when compared to `naive-accurate`). A trace is complete if there are no unrecoverable `rpcids` (explained in Section 4.3).

5.3.2 Results. For 2021, 58.32% of the traces have complete topology without needing to remedy any inconsistencies. Casper reconstructs an additional 25.5% of the traces, totaling to 83.82%.

For 2022, 86.42% of the traces have complete topology without needing to remedy any inconsistencies. Casper reconstructs an additional 12.18% of the traces, totaling to 98.6%.

6 DISCUSSION

Recommendation for consumers of the Alibaba datasets and related research papers: Users of the datasets should always specify their methodology for identifying and mitigating inconsistencies in their analyses. They should prefer the 2022 dataset, which contains fewer total inconsistencies. But, it is unclear if traces in the 2022 dataset were collected from the same applications or application versions as the 2021 dataset. The maximum trace sizes and maximum widths differ significantly between both years regardless of rebuild mode. As such, consumers may wish to use both datasets to test their work against a range of request-workflow characteristics.

We recommend caution when interpreting research that uses the 2021 dataset without specifying a methodology for handling inconsistencies. Readers should carefully consider if changes in trace characteristics or connectivity would affect the results. Of particular note is the Alibaba microservice analysis by Luo et al. [16]. This analysis uses a 7-day dataset of which the 2021 public release is a subset. But, does not specify whether the authors knew about the inconsistencies or whether they addressed them. As such, the presented graphs of trace characteristics, clustering results, and distributions suggest for the artificial trace generator may be suspect.

Exploring tradeoffs in capturing redundancies within trace data: Casper's functionality is possible because Alibaba's tracing infrastructure stores redundant data in caller/callee log messages. Namely, `rpcid` uniquely locates a call in the trace, allowing call-graph connectivity between services when intermediate calls are lost. But, `rpcids`' expressiveness results in larger context and larger network message sizes. Context-propagation errors can be circumvented by using chains of `UM` / `DM` fields. In contrast, the popular open-source model for distributed-tracing, OpenTelemetry [21], does not capture any redundancies. Spans (e.g., service executions) can be dropped if services' are too resource starved [22], leading to traces with (silent) missing nodes. Research is needed to explore how to encode redundancies in trace data or context and the overhead tradeoffs of doing so.

Tools to identify context-propagation errors: Capturing high-fidelity traces that represent their workflow relies on correct context propagation. Worse, context propagation errors can propagate downstream, making it difficult to identify the offending service. Tools, similar to lint, are needed that can detect whether services are propagating context correctly. These tools should be used prior to deploying new services or new versions.

7 SUMMARY

We systematized inconsistencies found in Alibaba's distributed tracing data and identified two root causes for these inconsistencies: data loss and context propagation errors. We built Casper, an toolkit which can remedy most inconsistencies in the trace data, building the largest accurate topologies. We evaluated Casper against other methods of constructing traces and show that our topologies are larger and more complex than other methods.

8 ACKNOWLEDGEMENTS

We thank the anonymous ACM/SPEC ICPE conference reviewers, Zhaoqi (Roy) Zhang, Chami Lamelas, Juraci Kröhling, Yuri Shkuro, and members of the D.O.C.C Lab at Tufts University for their insightful feedback and support. We thank the people at Alibaba who released such valuable data for our research community.

REFERENCES

[1] Alibaba's 2021 cluster trace of microservices 2021. https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2021.
[2] Alibaba's 2022 cluster trace of microservices 2022. https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2022.

[3] Vaastav Anand, Deepak Garg, Antoine Kaufmann, and Jonathan Mace. 2023. Blueprint: A Toolchain for Highly-Reconfigurable Microservice Applications. In *SOSP'23: Proceedings of the 25th Symposium on Operating Systems Principles.*

[4] BookInfo application 2024. https://istio.io/latest/docs/examples/bookinfo/

[5] Adrian Cockroft. 2016. The evolution of microservices. https://learning.acm.org/techtalks/microservices

[6] Confused of identifying services from MS_CallGraph_Table in MicroserviceTrace-v2021 2022. https://github.com/alibaba/clusterdata/issues/157.

[7] João Ferreira Loff, Daniel Porto, João Garcia, Jonathan Mace, and Rodrigo Rodrigues. 2023. Antipode: Enforcing Cross-Service Causal Consistency in Distributed Applications. In *SOSP'23: Proceedings of the 25th Symposium on Operating Systems Principles.*

[8] Martin Fowler. 2014. Microservices: a definition of this new architectural term. https://martinfowler.com/articles/microservices.html

[9] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 2021. Sage: practical and scalable ML-driven performance debugging in microservices. In *ASPLOS'21: Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems.* 135–151. https://doi.org/10.1145/3445814.3446700

[10] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *ASPLOS'19: Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems.*

[11] Github issue #157: Confused of identifying services from MS_CallGraph_Table in MicroserviceTrace-v2021 2021. https://github.com/alibaba/clusterdata/issues/157.

[12] Lexiang Huang and Timothy Zhu. 2021. tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces. In *SoCC'21: Proceedings of the 12th Symposium on Cloud Computing.* https://doi.org/10.1145/3472883.3486994

[13] Darby Huye, Yuri Shkuro, and Raja R. Sambasivan. 2023. Lifting the veil on Meta's microservice architecture: Aanalyses and topology and request workflows. In *ATC'23: Proceedings of the 2023 USENIX Annual Technical Conference.*

[14] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. 2017. Canopy: An end-to-end performance tracing and analysis system. In *SOSP'17: Proceedings of the 26th Symposium on Operating Systems Principles.*

[15] Chengzhi Lu, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu. 2023. Understanding and Optimizing Workloads for Unified Resource Management in Large Cloud Platforms. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23).* Association for Computing Machinery, New York, NY, USA, 416–432. https://doi.org/10.1145/3552326.3587437

[16] 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: Proceedings of the 12th Symposium on Cloud Computing.*

[17] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, Guodong Yang, and Chengzhong Xu. 2022. Erms: Efficient Resource Management for Shared Microservices with SLA Guarantees *(ASPLOS 2023).* Association for Computing Machinery, New York, NY, USA, 62–77. https://doi.org/10.1145/3567955.3567964

[18] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu. 2022. The Power of Prediction: Microservice Auto Scaling via Workload Learning. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco, California) *(SoCC '22).* Association for Computing Machinery, New York, NY, USA, 355–369. https://doi.org/10.1145/3542929.3563477

[19] Microservice MSCallGraph: a call is recorded more than twice 2022. https://github.com/alibaba/clusterdata/issues/124

[20] Sam Newman. 2021. *Building microservices: designing fine-grained systems* (2nd ed.). O'Reilly Media, Inc.

[21] OpenTelemetry 2024. https://opentelemetry.io/

[22] OpenTelemetry Memory Limiter Processor 2024. https://github.com/open-telemetry/opentelemetry-collector/blob/main/processor/memorylimiterprocessor/README.md

[23] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *OSDI'20: Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation.* 805–825.

[24] Korakit Seemakhupt, Brent E Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C Snoeren, Arvind Krishnamurthy, David E Culler, and Henry M Levy. 2023. A Cloud-Scale Characterization of Remote Procedure Calls. In *SOSP'23: Proceedings of the 25th Symposium on Operating Systems Principles.*

[25] Benjamin H. Sigelman, Luiz A. Barroso, Michael Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a large-scale distributed systems tracing infrastructure.* Technical Report dapper-2010-1. Google.

[26] Mert Toslali, Emre Ates, Alex Ellis, Zhaoqi Zhang, Darby Huye, Lan Liu, Samantha Puterman, Ayse K. Coskun, and Raja R. Sambasivan. 2021. Automating instrumentation choices for performance problems in distributed applications with VAIF. In *SoCC'21: Proceedings of the 12th Symposium on Cloud Computing.*

[27] Mert Toslali, Srinivasan Parthasarathy, Fabio Oliveira, Hai Huang, and Ayse K Coskun. 2021. Iter8: Online Experimentation in the Cloud. In *SoCC'21: Proceedings of the 12th Symposium on Cloud Computing.*

[28] Why are some call graph in cluster-trace-microservices-v2021 disconnected 2023. https://github.com/alibaba/clusterdata/issues/175.

[29] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G Edward Suh, and Christina Delimitrou. 2021. Sinan: ML-based and QoS-aware resource management for cloud microservices. In *ASPLOS'21: Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems.* 167–181. https://doi.org/10.1145/3445814.3446693

[30] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. 2018. POSTER: Benchmarking microservice Systems for software engineering research. In *ICSE'18 Companion: Proceedings of the 40th Companion to the International Conference on Software Engineering.*

[31] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, Xiongchun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, and Ratul Mahajan. 2022. Dissecting Service Mesh Overheads. arXiv:2207.00592 [cs.DC]