De-mystifying Microservice Topology Shape via a Deeper Study of the Alibaba Trace Dataset

Chami Lamelas
Tufts University

Linda Zhao
Tufts University

Abstract

Microservice design of cloud applications has been an area of academic research for some time. This has been motivated by more and more cloud applications being implemented as collections of lightweight microservices run on top of distributed systems. In this work, we seek to obtain a better understanding of Alibaba's microservice architecture via an analysis of the well-known Alibaba trace dataset. Our work provides some preliminary analysis of the architecture as well as identification of the various problems in the dataset that would need to be addressed in future research on this topic.

1 Introduction

More and more popular online cloud services are being implemented with lightweight microservices that are executed on a distributed system [7], typically using containers such as Docker [11]. Microservice-based applications have a variety of benefits such as implementation flexibility and an improved development process [9]. However, microservice-based designs are not without issues. On example is communication overhead. Researchers are interested in how the benefits and drawbacks of microservices are balanced in the design of cloud applications and how microservice-based implementations compare with monolithic implementations.

Academic research of microservice design has been impeded by the lack of publicly available, large scale applications implemented with microservices. This motivated the creation of DeathStarBench [7] an application suite that mimics common day-to-day cloud applications. However, later work has claimed that DeathStarBench is an unrealistic replacement for industry-scale microservice architectures. One such work is "Characterizing microservice dependency and performance: Alibaba trace analysis" by Luo et. al. from 2021 [10].

In this work, Luo et. al. released the Alibaba trace dataset [1] which contains traces from one of Alibaba's microservice-based applications. Luo et. al. conduct trace-level analysis in the interest of gaining insight into the microservice implementation of an industry-scale application. They do so by applying graph-learning techniques to traces and performing conventional clustering techniques to identify similar traces. Graph learning techniques have proven to be effective at capturing graph structure in fixed-dimensional embeddings in a variety of applications [16]. Due to the fact that the Alibaba trace dataset is collected from an industry-scale application, analysis upon it is considered to be more beneficial for understanding microservice-based cloud architectures.

Furthermore, the study of trace analysis is still an emerging field [2]. There exist a variety of works on how to collect important traces without producing high overhead [13] [5] [17]. However, there has been limited work in properly translating the immense amount of trace data collected by these tools into meaningful information both researchers and cloud engineers can learn from [2] [18]. Both are interested in possibilities for optimizing microservice design to improve system performance.

The observations expressed in the previous two paragraphs motivate our goal of extracting useful information from the Alibaba trace dataset. We analyze the Alibaba trace dataset at two levels: the microservice level and the trace level. For microservice-level analysis, we construct aggregated dependency graphs that group the microservice dependencies across the traces in the Alibaba dataset. We perform a statistical analysis of these graphs and discuss the difficulties with visualizing these graphs due to their immense scale.

For trace-level analysis, we construct call graphs corresponding to individual traces. These graphs contain the trees of calls that occur in the handling of requests. In the process of performing trace-level analysis, we discovered that the Alibaba trace dataset contains a wide variety of oddities and errors that make reproduction of Luo et. al.'s work [10] challenging. Hence, our work includes an in-depth discussion of the difficulty of utilizing the Alibaba trace data as well as our attempts to overcome some of the issues we discovered.

2 Background

The definition of a trace is somewhat vague. It is typically presented as a collection of communication events with some attached information, most critically, event-ordering information. In this section, we describe the Alibaba trace data as well as some terminology from the paper relevant to our work.

Traces in the Alibaba dataset are contained in the `MS_CallGraph_Table` table. The table contains data for over twenty million traces. It is split over 145 downloadable data files totaling over 203 GB. The table has the following schema. Note that from here on, we use communication event interchangeably with call and `MS_CallGraph_Table` row.

`timestamp` is the *signed* timestamp of a call. For some call types, a call with timestamp $-t$ ($t > 0$) corresponds to the response for call with timestamp in the interval $(t-\varepsilon, t+\varepsilon)$ for some small ε. Note ε is never formally specified in the `MS_CallGraph_Table` schema, we just use it to highlight that the response for a call with `timestamp` t will not necessarily be exactly $-t$.

`traceid` is the *unique* identifier for a trace. Calls matching on `traceid` are deemed to belong to the same trace.

`rpcid` is the *non-unique*, *hierarchical* identifier for a communication event. It is hierarchically similar to internet protocol addresses. We note here that despite being named as an identifier, `rpcid` is not unique. We address this in greater detail in sections 3 and 4.

`um` is the *upstream microservice* of the call. The *upstream microservice* calls the *downstream microservice* in a particular communication event.

`dm` is the *downstream microservice*, defined above.

`rpctype` The type of the communication. Examples include `http` and `rpc`. When `rpctype` is `http` or `rpc`, `timestamp` should be used to distinguish calls (positive `timestamp`) from responses (negative `timestamp`).

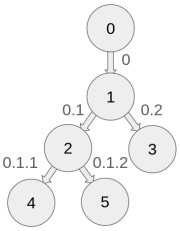
`roundtrip` is the round trip time of a call. This field is not used in our analysis. However, perhaps there is opportunity for `roundtrip` to be used in combination with positive `timestamp` values for the purpose of identifying concurrency in traces. Concurrency is different from width as discussed in section 5.

`interface` is the communication interface, to our knowledge it typically corresponds to a particular function or API. This field is also not used in our analysis.

For all future examples, in the interest of saving space, we shall omit the `roundtrip` and `interface` columns when showing example `MS_CallGraph_Table` rows. For the same reason, we will use integer `traceid` , `um` , and `dm` values in place of long randomized string values as in the raw data.

traceid	rpcid	um	dm	rpctype	timestamp
1	0	0	1	mc	1
1	0.1	1	2	mc	2
1	0.2	1	3	mc	3
1	0.1.1	2	4	mc	4
1	0.1.2	2	5	mc	5

(a) Some nice `MS_CallGraph_Table` rows.



(b) The corresponding call graph.

Figure 1: An example collection of nice rows from `MS_CallGraph_Table` and the corresponding call graph.

Given a collection of rows from `MS_CallGraph_Table` sharing the same `traceid` value, we wish to construct a *call graph*. An edge (u, v) in a call graph means that microservice u calls v in the trace. Microservices can appear multiple times at various points. Responses, as described above, are not included in call graphs. Figure 1a holds a collection of six `MS_CallGraph_Table` rows that belong to the same trace. The call graph corresponding to these rows is given in figure 1b. The `rpcid` for each row is used to place each edge in the call graph hierarchically. The call with `rpcid` 0 occurs before the call with `rpcid` 0.1 and so on. The *parent* call of a row with `rpcid` `x.y` has `rpcid` `x`. `um` and `dm` are used to identify the nodes on those edges. It is important to emphasize that we only consider *complete call graphs*. That is, call graphs that capture the communications that occur in an entire trace. It is definitely reasonable to collect partial call graphs, especially with all the issues that are present in the Alibaba dataset. We leave this to future work.

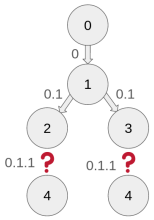
In our discussion of figure 1, we use the term "nice" to describe the rows in figure 1a because it is straightforward to construct the call graph corresponding to these rows. The majority of rows corresponding to an individual trace typically have a variety of oddities and errors that make it difficult to construct a call graph. This will be discussed in detail in section 3.

Lastly, we define an *aggregate dependency graph*. An edge (u, v) in an aggregate dependency graph denotes a relationship between microservice u and microservice v. Hence, microservices appear as unique nodes. Two particular types of aggregate dependency graphs will be described in section 4.

Once we have some call graphs constructed from the `MS_CallGraph_Table` rows, we are able to do further exploration and analysis over the trace through digging into the call graph structures. The forms of the reconstructed call graphs, as described in detail above, are tree-like directed graphs, where nodes are uniquely indexed microservices and the directed edges are invocations within each call, along with

traceid	rpcid	um	dm	rpctype	timestamp
1	0	0	1	mc	1
1	0.1	1	2	rpc	2
1	0.1	1	3	rpc	3
1	0.1.1	-1	4	mc	4

(a) A collection of `MS_CallGraph_Table` rows that make it impossible for one to construct a complete call graph.



(b) Ambiguity in call graph construction.

Figure 2: In this figure, we demonstrate with a simple example how the resulting ambiguity of differing `dm` values in duplicate `rpcid` value rows and missing `um` values can lead to the inability of constructing a complete call graph. Note that a `um` or `dm` value of -1 is used to denote a missing microservice for the purpose of conserving space.

the coordinating global `um` and `dm` values as node attributes. Considering that the topological structure of the call graphs might contain some features – indicating the behavior of and relationship among some microservices, we want to capture the graph structure features as graph embeddings. Feasible ways include traditional graph theories like kernel methods, or through graph neural networks (GNNs) [16], which use message passing techniques to propagate and aggregate the node information step by step within the graph structure. Referring to the Alibaba Trace Analysis paper, we chose to try reproducing the unsupervised GNN-based method in the paper as an initial attempt. As a result, our analyzing workflow is to first generate a graph-level embedding for each call graph, then do the dimensional reduction on each of them for "eigenvectors", and then try clustering over the root microservices. The detailed explanation can be found in section 4.

3 Problem

As described in the introduction, the goal of our work is to analyze the Alibaba trace dataset in the hope of learning some information about their microservice architecture. We perform microservice and trace-level analysis, as we will describe in detail in section 4. To perform these two types of analysis, there are a variety of aspects of the dataset that will need to be addressed beforehand. In this section, we describe the oddities and errors in the Alibaba trace dataset that hindered our ability to analyze it. Some of these issues will be handled in the following section. In this section, we make note of the issues we did not solve as well as the frequencies of some of the issues we identified.

In order to produce aggregate dependency graphs for the purpose of microservice-level analysis, one requires the presence of the `um` and `dm` fields in a row. However, some rows can have `"(?)"` or `NaN` values for these fields. According to the dataset GitHub repository [1], this is a side effect of trace collection errors. For our microservice-level analysis, we simply discarded any rows for which either of the `um` and `dm` fields is meaningless. Roughly 15% of around 100 billion rows were dropped.

There are a variety of oddities and errors in the dataset that make the construction of call graphs difficult and at times impossible. It is useful to note that this is an open area of discussion as evidenced by the eighty-four open issues on the dataset GitHub repository [1] at the time of writing.

The first oddity in the data is that rows belonging to the same trace are distributed across the data files. In fact, seven traces are split over 144 of the 145 data files. 98.6% of traces occur in one file or are split over contiguous files, meaning files that are numbered one after the other. Nearly 98% occur in one file. It is also important to note that the rows belonging to the same trace within a single file are not stored contiguously.

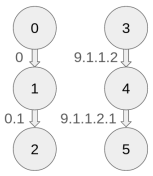
Another oddity in the data is the presence of duplicate `rpcid` values. That is, it is possible for multiple rows with the same `traceid` to have the same `rpcid` as well. Some instances of this can be attributed to the fact that we can have calls and responses for certain types of communication. This can be identified via negative `timestamp` values as discussed in section 2. However, we found that over 96% of over 20 million traces still had duplicate `rpcid` values after addressing the negative `timestamp` values.

Two more oddities stem from the duplicate `rpcid` oddity. The presence of duplicate `rpcid` rows with different `um` values and the presence of duplicate `rpcid` rows with different `dm` values. We discovered that 26% of traces have different `dm` values for rows with duplicate `rpcid` values. Note this is specifically different *non-missing* `dm` values. Recall the existence of missing `um` and `dm` values from the beginning of this section. We additionally discovered that 14% of traces have different `um` values for rows with duplicate `rpcid` values. These oddities can lead to one being unable to construct complete call graphs for certain traces. Figure 2 shows a toy example. That is, a collection of trace rows extracted from a larger, real trace. Since the fourth row of figure 2a has a missing `um` , we are not sure which microservice in the previous (0.1) level of the hierarchy called microservice 4 (it could have been 2 or 3). Thus, as illustrated in 2b, one cannot determine where to place the final edge in the call graph for the trace, leaving it incomplete.

Note that in the example trace rows shown thus far in figures 1 and 2, the top level `rpcid` value is 0. By top-level, we mean the `rpcid` value with the fewest delimiting periods. From here on we refer to this as the *root* `rpcid` value. Similarly, within a level of the hierarchy `rpcid` values increase by one in their rightmost *subfield*. That is, we see instances of 0.1, 0.2 or 0.1.1, 0.1.2. However, we found that it is possible that both of these conditions could be violated. For instance,

traceid	rpcid	um	dm	rpctype	timestamp
1	0	0	1	mc	1
1	0.1	1	2	rpc	2
1	9.1.1.2	3	4	rpc	3
1	9.1.1.2.1	4	5	rpc	4

(a) A collection of `MS_CallGraph_Table` rows that result in a call graph with multiple roots.



(b) Multiroot call graph.

Figure 3: In this figure we provide an example of `MS_CallGraph_Table` rows that translate to a call graph with multiple roots.

we found that 57% of traces contain a nonzero root `rpcid` value. In some number of traces, `rpcid` values belonging to the same level may begin with an `rpcid` whose rightmost subfield is not one, as with (the root) 9.1.1.2 in figure 3.

Up to this point in our discussion, one would assume that a trace has a single root `rpcid` value. This is motivated by the idea that a trace corresponds to a sequence of microservice communications that started from a root microservice that picked up the request into the overall system. However, we found that this is not always the case based on our understanding of `rpcid` values. We believe that `rpcid` values, while they may not have a prefix of 0, have a leftmost subfield that is some integer. This is motivated by the fact that there do not seem to be any instances of other potential root characters in the dataset such as an empty `rpcid` , which seems like a logical parent to `rpcid` value 0. Thus, a trace that has a row with `rpcid` 0 and a second row with `rpcid` 9.1.1.2 is considered to have two roots. Moving down (i.e. with increasing number of delimiting periods in `rpcid` values) in a row yields that the edges stemming from each of these roots do form trees as we originally expected the entire call graph to. Thus, instead of a call graph being a single tree, a call graph is a directed acyclic graph whose connected components are trees. This example is illustrated in figure 3. It is unclear what the cause of this behavior is, but perhaps it could be caused by some failure to properly update some internal counter within Alibaba's tracing system that is used to set the `rpcid` values throughout a trace.

One error we mentioned previously in the context of aggregated dependency graphs is the existence of `MS_CallGraph_Table` rows with missing `um` or `dm` values. In the context of trace-level analysis, 95% of traces are missing either an `um` or `dm` value in one of their corresponding rows. 45% of traces are missing both the `um` and `dm` values in one of their rows. We assume this is another symptom of programmer error when instrumenting the application.

Another error we came across was the possibility for traces to be missing `rpcid` values. For instance, a trace may have rows with `rpcid` values 0 and 0.1.1, but not a row with 0.1. In fact, 24% of traces are missing at least one `rpcid` level between a root and lower levels in the trace. Note that this does not account for the possibility of missing roots as the concept of a missing root is somewhat difficult to determine. Given the possibility for a root to start multiple `rpcid` levels down, it is unclear if any parents of such a root are missing. Consider again the example from figure 3. 9.1.1.2 starts three levels down based on delimiting periods. It is unclear whether it is missing any parents. Similarly, this does not consider the possibility of missing low call graph levels. For instance, it is possible that the example in figure 3 was meant to have rows on the same level as 0.1 or lower.

4 Approach

As described in the previous section, there are a variety of oddities and errors in the majority of `MS_CallGraph_Table` rows that hinder the call graph construction process shown in figure 1. In this section, we explain our preprocessing strategy as well as our microservice-level and trace-level analysis methods.

For our microservice-level analysis, we construct two types of aggregate dependency graphs. We define a *called by* graph as an aggregate dependency graph where an edge (u, v) denotes that microservice u is called by microservice v in at least one Alibaba trace. We define a *calling* graph as an aggregate dependency graph where an edge (u, v) denotes that microservice u calls microservice v in at least one Alibaba trace. In addition to constructing called by and calling graphs for the Alibaba traces, we also identify the number of traces a microservice m appears in. We construct these two graphs and identify microservice trace membership as follows. We applied statistical analysis to the two graphs and membership table, which is shown in section 5.

1. Preprocess the data by dropping all rows from the data files where either the `um` or `dm` value is missing.

2. Construct the called by graph by, for each m, identifying the set of microservices that appear as the `um` value of a row where m is the `dm` value.

3. Construct the calling graph by, for each m, identifying the set of microservices that appear as the `dm` value of a row where m is the `um` value.

4. Construct microservice trace membership table by identifying the set of `traceid` values where microservice m appears as either the `um` or `dm` value in a `MS_CallGraph_Table` row with said `traceid` value.

For our trace-level analysis, we had to construct call graphs from traces after preprocessing the traces to address as many of the oddities and errors from section 3 as possible. We first describe our preprocessing strategy for a trace t represented as a collection of `MS_CallGraph_Table` rows.

traceid	rpcid	um	dm	rpctype	timestamp
1	0	0	1	mc	1
1	0.1.1	2	3	rpc	3

traceid	rpcid	um	dm	rpctype	timestamp
1	0	0	1	mc	1
1	0.1	-2	-3	sub	0
1	0.1.1	2	3	rpc	3

Figure 4: In this figure we provide an example of how missing `rpcid` values are corrected using our approach. We identify 0 as a root and that there should be a 0.1 row between 0 and its sublevel 0.1.1. A blank row is then substituted into the trace's rows. Note that we use -2 and -3 to mark placeholder `um` and `dm` values in the inserted artificial row to conserve space.

1. Reduce the frequency of duplicate `rpcid` values in t by dropping all rows of the *minority sign* for a particular `rpcid` value given the `rpctype` is `rpc` or `http`. That is, for a particular `rpcid` suppose the count of rows with positive `timestamp` values is p and the count of rows with negative values is n. We keep the p positive rows if $p > n$ else we keep the n negative rows. While this approach is somewhat blind, there is no way to match up negative and positive `timestamp` rows due to the existence of ε as described in section 2. Note, if we ever encounter rows with duplicate `rpcid` values that have an `rpctype` other than `rpc` or `http`, we drop the duplicate rows. This is an instance of user error, as noted on the dataset repository [1].

2. Fill in missing `rpcid` levels in t by identifying the root `rpcid` values of t and then ensuring that any rows with non-root `rpcid` values can reach the roots. We ensure that all rows with non-root `rpcid` values can reach roots by substituting rows with the necessary `rpcid` values. We improve the runtime of this approach by maintaining a table of the `rpcid` values in t that we have already determined to reach the root to avoid unnecessary recursion. Figure 4 provides an example of this. First, we identify that 0 is the single root `rpcid` value and 0.1.1 is the single non-root `rpcid` value. To make 0.1.1 reach 0, we need to add the parent for 0.1.1, i.e. 0.1. Since t is a set of rows, we need to substitute in a row with placeholder values. Our placeholder values are -2 for `um` , -3 for `dm` , `sub` for `rpctype` , and 0 for `timestamp` . For the purposes of our analysis, our `rpctype` and `timestamp` placeholders have no effect. However, the insertion of effectively meaningless `um` and `dm` values creates the potential for additional traces missing microservice values.

traceid	rpcid	um	dm	rpctype	timestamp
1	0	0	-1	mc	1
1	0.1	1	2	rpc	2
1	0.1.1	-1	3	rpc	3

um	dm
0	1
1	2
2	3

Figure 5: In this figure we provide an example of how our approach attempts to patch missing microservice identifiers. We identify the first row's `dm` based on the second row's `um` and the third row's `um` based on the second row's `dm` . We only show the corrected `um` and `dm` values in the three rows to conserve space as no other fields in these rows are modified in this preprocessing stage.

3. Reduce the frequency of missing microservices by making use of the assumption that the parent row `dm` value should match the child row's `um` value. Note that parent and child are determined based on `rpcid` as described in section 2. Figure 5 shows an example. The row with `rpcid` 0 is missing its `dm` value. Hence, we use the `um` value of the row with a child `rpcid` (0.1) to patch the missing value. The row with `rpcid` 0.1.1 is missing its `um` value. Hence, we use the `dm` value of the row with the parent `rpcid` (0.1) to patch the missing value. We can apply this same strategy when we have `um` or `dm` being -2 and -3 as well, hence this preprocessing step must succeed step 2.

4. Drop t if it may be impossible to derive the call graph by identifying if there are rows with duplicate `rpcid` values but differing `um` or `dm` values. This is done to the avoid microservice ambiguity shown in figure 2. It is possible that in some instances, a complete call graph could be constructed, but for the purpose of simplicity, we avoid this issue. We address a potential improvement in section 6. It is important to note that 34% of 20 million traces are dropped because of this.

Figure 6 provides an example of our preprocessing strategy on a simplified Alibaba trace. First, we identify that the minority sign for `rpcid` 0.1.1 is positive hence the 0.1.1 row with `timestamp` -3 is dropped. Second, we identify that 0.1.1 and its descendant rows cannot reach their corresponding root of 0. Hence, we add a row with placeholder values. Lastly, we patch three missing microservices. Figure 7 shows the resulting trace rows after preprocessing. It is important to note that there are still some missing microservices in the trace. In fact, it is almost always the case that our missing microservice patching strategy fails to remove all missing microservices. We will return to this figure after we describe our call graph

rpcid	um	dm	rpctype	timestamp
9.1.1.1	-1	-1	mc	2
0.1.1	2	3	rpc	3
0	-1	1	rpc	1
0.1.1	2	3	rpc	3
0.1.1.1	3	4	rpc	4
9.1.1	5	6	mc	1
0.1.1	2	3	rpc	-3

1. Delete

0.1.1	2	3	rpc	-3

2. Insert

0.1	-2	-3	sub	0

3. Replace

9.1.1.1	6	-1	mc	2
0.1	1	2	sub	0

Figure 6: This figure illustrates an example of how our preprocessing approach affects `MS_CallGraph_Table` rows via an insertion, deletion, and replacement.

construction strategy. Below is our call graph construction process expressed in Python.

```
1  def callgraph(trace):
2      trace = preprocess(trace)
3      if trace is None:
4          return None
5      ROOT_PREFIX = "~"
6      graph = defaultdict(list)
7      roots = set(ct.find_roots(trace))
8      rpcdups = Counter()
9      microservices = dict()
10     for i, row in enumerate(trace):
11         if i in roots:
12             rpckey = ROOT_PREFIX + str(i)
13             microservices[rpckey] = c.um(row)
14         else:
15             rpckey = ct.get_parent(c.rpc(row))
16         newrpc = c.rpc(row)
17         if rpcdups[newrpc] > 0:
18             newrpc += f"_{rpcdups[newrpc]}"
19         rpcdups[c.rpc(row)] += 1
20         microservices[newrpc] = c.dm(row)
21         graph[rpckey].append(newrpc)
22     indexing = {k: i for i, k in
23                 enumerate(microservices)}
24     edgelist = [(indexing[k], indexing[v])
25                 for k, adj in graph.items()
26                 for v in adj]
27     return edgelist,
28            list(microservices.values())
```

In summary, this code builds up the dictionary `graph` mapping parent `rpcid` values to child `rpcid` values to emulate an adjacency list. When parent `rpcid` values are not present (i.e. for roots) we use a special key (lines 11-12). When there are duplicate child `rpcid` values, we amend the `rpcid` values in a way so that they are unique (lines 16-18) so we maintain the correct number of duplicate `rpcid` value children of a parent. The call graph is represented as an edge list specified by unique node identifiers $0, 1, 2, \ldots$ which IS built from `graph` (lines 24-26). This code additionally provides the microservices that correspond to those node identifiers in the edge list (see `microservices`).

rpcid	um	dm	rpctype	tstmp
9.1.1.1	6	-1	mc	2
0.1.1	2	3	rpc	3
0	-1	1	rpc	1
0.1.1	2	3	rpc	3
0.1.1.1	3	4	rpc	4
9.1.1	5	6	mc	1
0.1	1	2	sub	0

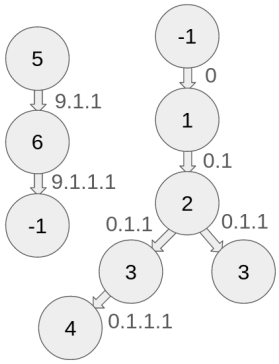


Figure 7: This figure displays the resulting rows after preprocessing as well as the corresponding call graph. Observe most importantly that microservices may still be missing.

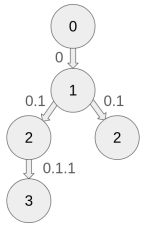
There is an important side effect of line 21. That is, when we have duplicate `rpcid` values, all descendant `rpcid` rows are attached to one particular node in the corresponding call graph. An example of this is provided in figure 8. Here there are duplicate `rpcid` values of 0.1 and any instance of descendants (0.1.1) will be added to the first of them. As a result, the trees extending from each root of our constructed call graphs will always be left heavy.

It is important to note that both our preprocessing and call graph construction strategies can be easily parallelized assuming the rows of a trace are contained on a single file. Recall from section 3 that this is not the case for 2% of traces. The combined preprocessing and call graph construction operations are quite time intensive. In order to get results in a more reasonable time frame, we dropped all traces that are split over multiple files. This enabled us to implement a primitive version of the mapping stage of the MapReduce [3] paradigm where traces on a data file are preprocessed and transformed into call graphs independently from the traces on the other 144 data files. Note this parallelism is realistically limited by the number of available CPU cores. In practice, with this parallelism optimization, the preprocessing and transformation took around four hours on the c220g5 type node on CloudLab [4].

In terms of the graph learning-based trace analysis, we originally attempted to do as the Alibaba Trace Analysis paper suggests, using the *InfoGraph* [14] to calculate graph-level embeddings for each call graph, then doing dimensional reduction on them all to obtain 2-d plots, trying to investigate some similar patterns and clustering over the traces and microservices. However, the source code provided by *InfoGraph*

traceid	rpcid	um	dm	rpctype	timestamp
1	0	0	1	mc	1
1	0.1	1	2	rpc	2
1	0.1	1	2	rpc	3
1	0.1.1	2	3	mc	4

(a) A collection of `MS_CallGraph_Table` rows where there are duplicate `rpcid` values, but identical `dm` values.



(b) The call graph produced by our call graph construction strategy.

Figure 8: In this figure we provide an example of duplicate `rpcid` values that is handled by our call graph construction strategy. Note that our strategy will construct call graph trees that will always be left heavy.

does not seem consistent with its description, as providing an unsupervised graph embedding model through contrastive learning on graph structures. Therefore, we use *Graph2Vec* as an alternative, and the workflow of our GNN-based analysis is as follows:

Calculate graph-level embeddings in the form of a 20-dimensional vector for each call graph using *Graph2Vec*.

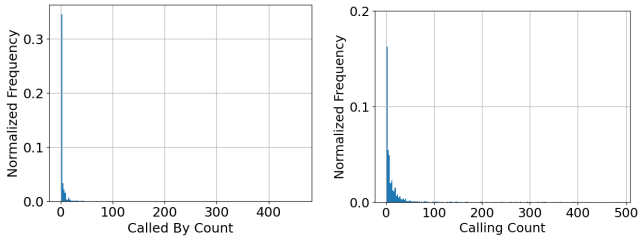
Calculated 2-D eigenvector from graph embeddings using Principle Components Analysis (PCA) on each call graph, which extracts the two most weighted orthogonal eigenvectors within the embedding vector space.

Plot and Analyze the eigenvectors. On the trace level, we plot all eigenvectors within the same figure. On the microservice level, we aggregate the call graphs based on the root microservice and separately plot eigenvectors for each root microservice, as well as put all eigenvectors together and color them according to the coordinating root microservice.

As a result of this procedure, we generate an eigenvector plot for each root microservice, along with several plots containing all eigenvectors, shown in section 5.

5 Results

In this section, we describe some of the results from our microservice-level and trace-level analysis. Using our approach from section 4 for constructing called by and calling graphs, we discovered that there are nearly 17,000 unique microservices and around 56,000 unique microservice dependencies in the 85% of rows that were used in constructing the aggregate dependency graphs.



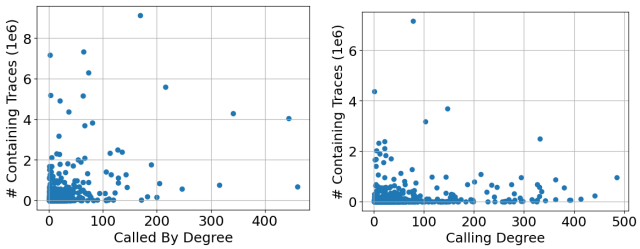
(a) The called by degree distribution. (b) The calling degree distribution.

Figure 9: In this figure we provide the degree distributions of the called by and calling aggregate dependency graphs of the Alibaba dataset.

Figure 9 provides the *degree distributions* of the called by and calling graphs. We define the degree distribution as a frequency histogram of the number of outgoing edges per node in an aggregated dependency graph. Figure 9 demonstrates that the degree distributions of the called by and calling graphs follow a power law distribution, similar to social networks [12]. Thus, figure 9a illustrates that the majority of microservices are called by only one microservice over all the Alibaba traces. The most popular microservice is called by 460 microservices. Similarly, figure 9b demonstrates that the majority of microservices call only one microservice. The maximum calling microservice calls 485 microservices.

These observations motivate the question: why not combine microservices that only call one microservice? It is important to note that microservices can call themselves in `MS_CallGraph_Table` rows and that the Alibaba trace dataset is from a small segment of time (12 hours) [1]. This can mean that microservices are spawning some parallel work to complete some task. However, the idea of splitting up even one part of a microservice's task into another microservice could be language or environmental differences. It is known that microservice-based design enables implementing specific components of an application in a particular language and environment [9].

We collected some additional aggregate dependency graph statistics as we were unable to successfully visualize these graphs due to their large number of nodes and edges. We found that many NetworkX [8] samples were applied to graphs with low thousands of nodes and hundreds of edges. The aggregate dependency graphs are extremely sparse with a sparsity ratio of $2 \cdot 10^{-4}$. We define the sparsity ratio as the ratio of the present edges in a graph G divided by N^2 where N is the number of nodes in G. Additionally, the vast majority of microservices are "related". That is, 99% of microservices are connected. It is possible that the unrelated microservices that are contained in small disconnected components have been detached due to the removal of missing `um` or `dm` value rows.



(a) The correlation between trace frequency and called by degree.

(b) The correlation between trace frequency and calling degree.

Figure 10: In this figure we provide the correlation between microservice usage and trace frequency. The trace frequencies are given in millions.

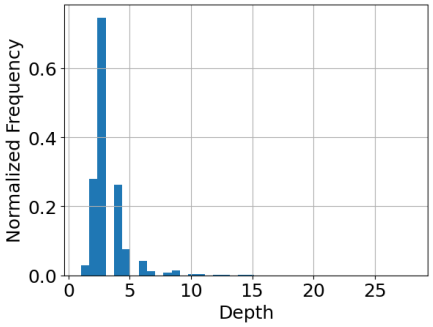


Figure 11: In this figure we provide the depth distribution of the traces for which we constructed call graphs using the approach in section 4.

Figure 10 plots microservice trace frequency with their microservice popularity (10a) and usage (10b). These figures demonstrate that there is no obvious correlation between trace frequency and how many microservices a microservice uses or is used by. The trace frequency of microservice m is the number of traces for which m occurs as either the `um` or `dm` of one of its rows. The Pearson correlation coefficient for figure 10a is $r = 0.4378$. The coefficient for figure 10b is $r = 0.2184$. These values support our claim of low correlation. Most microservices occur in a small set of traces. 94% of microservices appear in less than 10,000 (0.05%) of traces. One microservice appears in over nine million traces.

We now move to our trace-level analysis. Figure 11 provides the distribution of *trace depth* in the call graphs constructed using the approach from section 4. We define trace depth as the maximum height of the trees in a trace's call graph. We use the traditional definition of tree height as the longest root-to-leaf path in the tree. This can be calculated efficiently for a trace t by calculating the difference between the maximum number of delimiting periods in a trace's `rpcid` values and the minimum number of periods. Figure 11 indicates that most traces are not too deep, the maximum trace depth being 28.

We additionally analyzed *trace width*. We define trace width as the maximum of the largest level sizes for each tree in a trace t's call graph. This is computed efficiently using the fact that, for a tree in t's call graph, the rows of t with the same number of delimiting periods in their `rpcid` values belong to the same level. Most traces are not very wide, 85% of considered traces have a width less than ten. It is important to note that width is not the same as concurrency.

As for the GNN-based analysis, we separately generated plots for each root microservice, in the hope of going over them all and finding some general patterns. In practice, we did discover some common patterns, shown in Fig 16. We also plot all eigenvectors together in Fig 12 to explore some trace-level patterns, colored by the corresponding root microservice. The more in-depth descriptions are below the figures.

6 Contributions

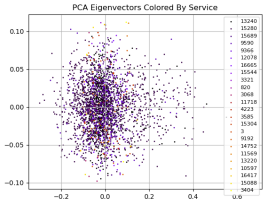
Midway through our work, we were unable to construct call graphs from the raw trace data due to the oddities described in the previous sections. However, we believed we would still be able to construct some traces partially. This was indeed the case as described above. We believe that there is an opportunity to partially construct even more traces using the following general strategy:

Preprocess the raw trace data in a similar manner to the strategy we describe in section 4.

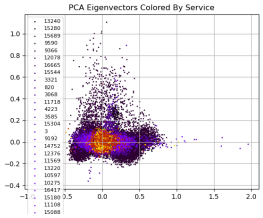
Construct the call graph by initially relying purely on `rpcid` values (like the Python program in section 4) but utilize `um` and `dm` values where possible to handle the duplicate `rpcid` differing `dm` problem.

The primary goal we hoped to accomplish given the ability to construct call graphs from traces was to compute fixed-dimension vector embeddings from the graphs. We were able to do this as described above using *Graph2Vec*. From there, we were able to accomplish our additional goal of constructing eigentraces as influenced by the concept of eigenfaces [15] as well as being able to display these embeddings in low dimensional space, in an attempt of finding similar microservices depending on their eigenvector patterns.

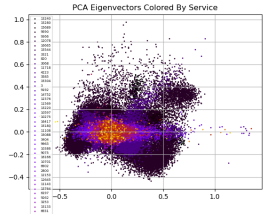
Another goal we hoped to accomplish was to find a way to better visualize aggregate dependency graphs to convey useful information about microservice relationships across traces. However, we were unable to do so despite lengthy investigations of NetworkX [8] and Graphia [6], mostly due to the too-large scale of the dataset. We are less optimistic about future improvements in this area. The death star diagram of microservice dependencies has been shown in existing work [7]. We find this diagram not particularly useful, but



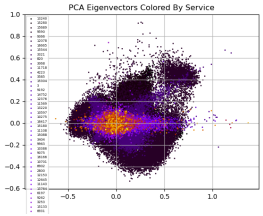
(a) Eigenvectors plot after processing 1 call graph file.



(b) Eigenvectors plot after processing 50 call graph file.

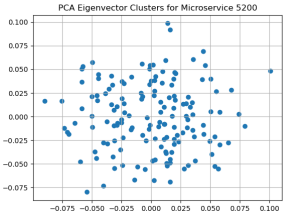


(c) Eigenvectors plot after processing 100 call graph file.

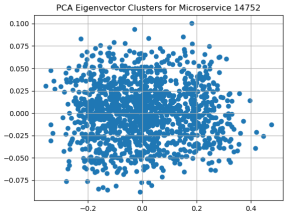


(d) Eigenvectors plot after processing 145 call graph file.

Figure 12: In this figure we present four plots of all eigenvectors, generated from different numbers of files, from 1 to 145 in total. Each point is coordinated with one eigenvector, or a trace, colored by the root service. Although the plots seem messy, we could combine them with the eigenvector plots for single root services, and infer that the shape is an overlay of several common microservice patterns as a whole.

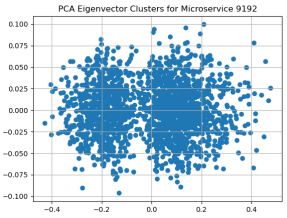


(a)

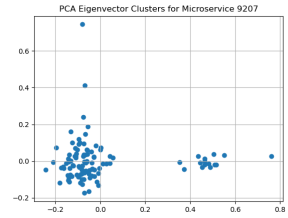


(b)

Figure 13: Some eigenvector plots appears to form one single cluster in the center

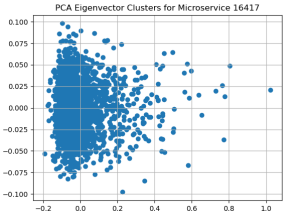


(a)

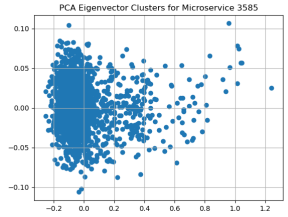


(b)

Figure 14: Some eigenvector plots appears to form two clusters on the left and right respectively.

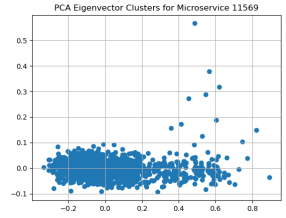


(a)

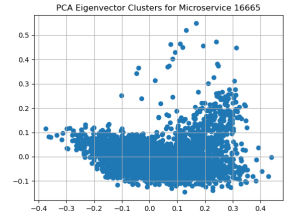


(b)

Figure 15: Some eigenvector plots have special patterns like one vertical cluster on the left spreading rightward.



(a)



(b)

Figure 16: Some eigenvector plots have special patterns like one horizontal cluster at the bottom spreading upwards.

were not able to produce any plots that yielded more useful information. This includes attempts to weight graph visualizations based on various metrics such as microservice usage.

7 Conclusions

In this work, we identify a range of oddities and errors in the Alibaba trace dataset, describe our approach to handling a subset of these issues, and perform some analysis of the dataset on both the microservice and trace levels. While we are able to address some of the issues in the dataset, they make subsequent analysis quite difficult. From our analysis of the Alibaba trace dataset, we learned that microservice usage tends to follow a power law distribution but is not correlated with trace occurrence. Traces tend to be neither wide nor deep.

We believe that, in future work, some more issues could be resolved, and trace analysis would be far more effective on a better-structured dataset. For example, we did GNN-based analysis only on the microservice level, which could be extended to the trace level as well. Moreover, we applied the *Graph2Vec* model to extract call graph embeddings here, which generally achieves substructure representations for a graph. However, it is mostly hard to specifically explain the meaning of the embeddings from a trace or microservice level, i.e. what exact information of the call graphs is captured and how it is encoded in the graph embeddings. Knowing that, we would be able to better extract some patterns and regulations within the call graphs, and combine them with microservices', and even the system's behaviors. This problem could possibly be improved by refining the input (e.g. using a better dataset or including more related attributes as input) and adjusting the GNN design.

Acknowledgments

This work was completed as part of the spring 2023 offering of Tufts University course COMP 150 Debugging Cloud Computing taught by Dr. Raja Sambasivan. We thank Dr. Sambasivan and Tufts University computer science Ph.D. candidate Darby Huye for their inspiration and advice for this project.

Availability

Our code is publicly available at this GitHub repository.

References

[1] Alibaba 2021 trace dataset. https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2021. Accessed: 2023-05-05.

[2] Thomas Davidson, Emily Wall, and Jonathan Mace. A qualitative interview study of distributed tracing visualisation: A characterisation of challenges and opportunities. *IEEE Transactions on Visualization and Computer Graphics*, 2023.

[3] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[4] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[5] Rodrigo Fonseca, George Porter, Randy H Katz, and Scott Shenker. X-trace: A pervasive network tracing framework. In *4th {USENIX} Symposium on Networked Systems Design & Implementation ({NSDI} 07)*, 2007.

[6] Tom C. Freeman, Sebastian Horsewell, Anirudh Patir, Josh Harling-Lee, Tim Regan, Barbara B. Shih, James Prendergast, David A. Hume, and Tim Angus. Graphia: A platform for the graph-based visualisation and analysis of high dimensional data. *PLOS Computational Biology*, 18(7):1–17, 07 2022.

[7] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[8] Aric Hagberg and Drew Conway. Networkx: Network analysis with python. *URL: https://networkx. github. io*, 2020.

[9] Pooyan Jamshidi, Claus Pahl, Nabor C Mendonça, James Lewis, and Stefan Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35, 2018.

[10] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 412–426, 2021.

[11] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), mar 2014.

[12] Tushti Rastogi. A power law approach to estimating fake social network accounts. *arXiv preprint arXiv:1605.07984*, 2016.

[13] Benjamin H Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. 2010.

[14] Fan-Yun Sun, Jordan Hoffmann, Vikas Verma, and Jian Tang. Infograph: Unsupervised and semi-supervised graph-level representation learning via mutual information maximization. *arXiv preprint arXiv:1908.01000*, 2019.

[15] Matthew Turk and Alex Pentland. Eigenfaces for recognition. *Journal of cognitive neuroscience*, 3(1):71–86, 1991.

[16] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 32(1):4–24, 2020.

[17] Lei Zhang, Zhiqiang Xie, Vaastav Anand, Ymir Vigfusson, and Jonathan Mace. The benefit of hindsight: Tracing {Edge-Cases} in distributed systems. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 321–339, 2023.

[18] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. {CRISP}: Critical path analysis of {Large-Scale} microservice architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 655–672, 2022.