Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis

Shutian Luo*‡
Shenzhen Institute of Advanced Technology, CAS
Univ. of CAS, Univ. of Macau
st.luo@siat.ac.cn

Huanle Xu*‡
University of Macau
huanlexu@um.edu.mo

Chengzhi Lu‡
Shenzhen Institute of Advanced Technology, CAS
Univ. of CAS
cz.lu@siat.ac.cn

Kejiang Ye‡
Shenzhen Institute of Advanced Technology, CAS
kj.ye@siat.ac.cn

Guoyao Xu
Alibaba Group
yao.xgy@alibaba-inc.com

Liping Zhang
Alibaba Group
liping.z@alibaba-inc.com

Yu Ding
Alibaba Group
shutong.dy@alibaba-inc.com

Jian He
Alibaba Group
jian.h@alibaba-inc.com

Chengzhong Xu†‡
University of Macau
czxu@um.edu.mo

ABSTRACT

Loosely-coupled and light-weight microservices running in containers are replacing monolithic applications gradually. Understanding the characteristics of microservices is critical to make good use of microservice architectures. However, there is no comprehensive study about microservice and its related systems in production environments so far. In this paper, we present a solid analysis of large-scale deployments of microservices at Alibaba clusters. Our study focuses on the characterization of microservice dependency as well as its runtime performance. We conduct an in-depth anatomy of microservice call graphs to quantify the difference between them and traditional DAGs of data-parallel jobs. In particular, we observe that microservice call graphs are heavy-tail distributed and their topology is similar to a tree and moreover, many microservices are hot-spots. We reveal three types of meaningful call dependency that can be utilized to optimize microservice designs. Our investigation on microservice runtime performance indicates most microservices are much more sensitive to CPU interference than memory interference. To synthesize more representative microservice traces, we build a mathematical model to simulate call graphs. Experimental results demonstrate our model can well preserve those graph properties observed from Alibaba traces.

ACM Reference Format:
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 *ACM Symposium on Cloud Computing (SoCC '21), November 1–4, 2021, Seattle, WA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3472883.3487003

1 INTRODUCTION

Loosely-coupled and light-weight microservices are gradually replacing monolithic applications [4]. Comparing with previous monolithic applications, microservices have special advantages in many aspects, such as cross-team development, friendly deployment, and so on. Nowadays, leading cloud service companies such as AWS and Alibaba start to provide an off-the-shelf microservices architecture for users to ease their application deployment [1, 3, 5].

To leverage microservice architecture, many benchmarks such as Acme Air [33], μSuite [27], and DeathStarBench [16] have been developed to explore the major characteristics of microservices. These benchmarks compare microservices and monolithic applications with respect to network overhead [33], remote procedure calls efficiency [27], and performance implications on resource management and hardware

*Co-first author. Both authors contributed equally to this paper.
†Corresponding author.
‡S. Luo, H. Xu, C. Lu, K. Ye and C. Xu are also with Guangdong-Hong Kong-Macao Joint Laboratory of Human-Machine Intelligence-Synergy Systems.

SoCC '21, November 1–4, 2021, Seattle, WA, USA

ACM ISBN 978-1-4503-8638-8/21/11.
https://doi.org/10.1145/3472883.3487003

[16]. However, these studies only provide insights into relatively small-scale clusters. The results do not necessarily apply to production environments.

In this paper, we aim to make a comprehensive analysis of the large-scale deployment of microservices in production clusters. Specifically, we analyze the behaviors of more than 20,000 microservices in a 7-day period and profile their characteristics, including anatomy of dynamic call graphs and characterization of microservice dependency as well as the runtime performance analysis. More importantly, we build a stochastic model to simulate the dynamic dependencies between microservices. We illustrate our studies and the important findings in the following.

Microservice call graphs are substantially different from traditional DAGs of data-parallel jobs. Although a microservice call graph can be viewed as a direct graph, it presents several distinct features as listed below.

- The size of microservice call graphs follows a heavy-tail distribution. Around 10% of call graphs consist of more than 40 microservice stages, whereas most data-parallel jobs only contain a few stages.
- A microservice call graph is topologically similar to a tree. In Alibaba microservice traces, a majority of nodes have in-degrees of one. By contrast, observations from big data job traces imply traditional DAGs usually contain multiple gather components [31].
- A non-negligible fraction of microservices are hot-spots. Specifically, about 5% of microservices are multiplexed by more than 90% of online services in Alibaba clusters. However, traditional DAG graphs do not share nodes with each other.
- Microservices can form highly dynamic call dependencies in runtime. In an extreme case, the same online service can have more than nine classes of topologically different graphs. As a comparison, traditional DAG graphs tend to be static and do not change after a job is submitted.

Strong dependency between microservices provides good opportunities for optimization of microservice designs. A pair of two microservices can form a strong cyclic dependency: The interface of an incoming call to upstream microservice (UM) is the same as the reply interface of UM for downstream microservice (DM) to call back. In this case, coupling these two interfaces together can potentially avoid unnecessary deadlocks. Moreover, a noticeable fraction of microservice pairs have a strong coupled dependency. For such pairs, the UM will repeatedly call the DM multiple times whenever the UM is called by others. Coupling these interfaces with a strong dependency may help to greatly reduce the communication overhead.

Microservice performance is much more sensitive to CPU interference than memory interference. Microservices in Alibaba clusters are usually deployed with hundreds of containers, which are co-located with batch applications on multiple physical hosts. The resource utilization of microservice containers can be as low as 10% usually, and the resulted response time (RT) does not vary much across different workloads. However, CPU interference can greatly hurt RT performance. In particular, a host CPU utilization of 30% can degrade the average RT by 20% when compared to utilization of 10%. As a consequence, there is a strong demand for more efficient job schedulers that can well balance the CPU utilization across different hosts. In addition, we reveal that the RTs of microservices are highly graph topology dependent.

Stochastic models can simulate dynamic microservice call graphs quite well. Existing microservice benchmarks have a very small scale since the number of microservices does not exceed 40, e.g., [16, 27, 43]. Another great limitation of these benchmarks is that, even though they contain multiple online services but each service keeps a fixed call graph topology that does not change often when different requests are being processed. To mitigate these limitations and generate microservice traces on a much larger scale, we build a stochastic model to simulate microservice call graphs aiming at preserving graph properties observed from Alibaba traces. And this model relies on a good classification of different types of microservices to generate graphs. The experimental results demonstrate our graph simulator can well preserve these graph properties.

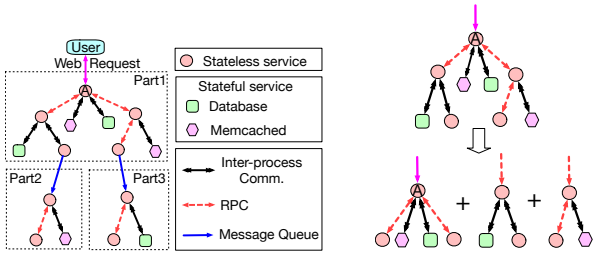
To summarize, we have made the following contributions in this paper:

- We conduct the first comprehensive study on large-scale deployment of microservices in production clusters. Our study covers both the structural properties of microservice call graphs (§ 3) as well as microservice call dependencies (§ 4).
- We make a thorough characterization of microservice runtime performance, which provides deep insights into microservice scheduling and resource management. (§ 5)
- We build a graph model to efficiently generate microservice traces on a large-scale. More importantly, we also conduct a theoretical analysis to characterize structural properties in simulated graphs. (§ 6)

2 MICROSERVICE BACKGROUND AND ALIBABA TRACE OVERVIEW

2.1 Microservices architecture

2.1.1 Call graph. A microservice usually runs in multiple containers to serve users' requests. The request from users is called an *origin* request and this request is first sent to an *Entering Microservice*, which then triggers a series of calls between related microservices. We define the set of these



(a) Components of call graphs **(b)** Two-tier invocations

Figure 1: Illustrations of microservices.

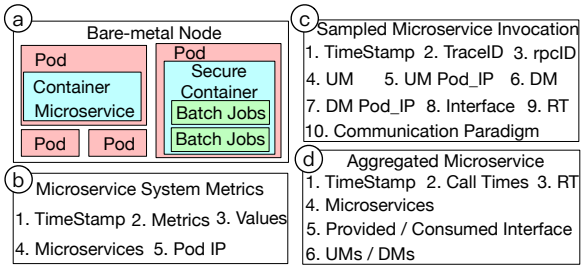


Figure 2: Alibaba microservice traces information.

calls as a call graph. Thus, a call graph contains multiple calls between different pairs of microservices. Here, a pair of microservices contains one upstream microservice (UM) and one downstream microservice (DM). For ease of illustration, we sketch a call graph as shown in Fig. 1(a). User issues an origin web request via HTTP to *Entering Microservice* A which is a front-end web service. When replying to the *origin* request, Microservice A shall call its DMs in turn further call their downstream microservices.

Microservices can be categorized into two types, namely, stateless (a circle in Fig. 1(a)) and stateful (a rectangle or hexagon). Stateless services are isolated from state data while stateful services such as databases [8] and Memcached [13] need to store data. Stateful services often provide a small number of uniform query interfaces such as reading or writing data, while stateless services tend to provide tens to hundreds of evolving interfaces for different purposes.

There exist three types of communication paradigms between a pair of microservices, i.e., inter-process communication [30], remote invocation [28], and indirect communication [16]. Inter-process communication (IP) usually happens between stateless and stateful microservices. Remote invocation such as Remote Procedure Call (RPC [30]) is a two-way communication under which a DM must return a result to its corresponding UM. By contrast, indirect communication such as Message Queue (MQ) is one-way only [30]. Under such communications, the UM sends a message to the third entity, which will persist the message for reliability, and the DM fetches the message on demand from the third entity directly without a reply. Remote invocation has a high efficiency while indirect communication maintains good flexibility.

2.1.2 Hierarchical call dependencies and RTs. As shown in Fig. 1(a), the call graph can be divided into several *parts* according to the edge of indirect communication. Each *part* can consist of multiple two-tier invocations with each consisting of a UM and all the DMs it calls (Fig. 1(b)). The *call depth* (aka the number of tiers) is defined as the length of the longest path in a call graph, e.g., the call depth of the graph illustrated in Fig. 1(a) is 5 (or it has 5 tiers).

The response time (RT) of a call is the length of the interval from UM calling its DM to it receiving the response. Since an indirect communication does not need to return a result, RT of an *origin* request is dominated by the part associated with its user (e.g., Part 1 in Fig. 1(a)). The same class of user requests can trigger different microservice call procedures and thus incur heterogeneous RTs. Take online ordering for an example, depending on whether a user holds a coupon ticket or whether there is a sale, the set of two-tier invocations involved in the ordering is quite a case dependent, so is the time to complete this ordering.

2.2 Alibaba trace overview

In this paper, we analyze more than ten billion call traces among nearly twenty thousand microservices in 7 days from Alibaba cluster. We believe such a scale of samples can well represent the wide deployment of all microservices in Alibaba [1]. In the following, we shall give a detailed description of these traces.

2.2.1 Physical running environment. Alibaba clusters adopt Kubernetes [7] to manage the bare-metal cloud [39] and, relies on the hardware-software hybrid virtio I/O system to enhance cluster performance and achieve better isolation between different services. As shown in Fig. 2(a), online services and offline jobs usually coexist in the same bare-metal node, so as to increase the cluster resource utilization. Online services (e.g., microservices) are running in containers, which are managed by Kubernetes directly. By contrast, for batch jobs, Kubernetes shall first allocate a certain amount of pods (the basic resource scheduling unit under Kubernetes [9]), which will then be delivered to Fuxi [41], a scheduler for batch jobs in Alibaba, for further scheduling. Each of these pods will run in a secure container [6, 10]to process batch jobs. The adoption of bare-metal servers and deployment of secure containers could alleviate the impact of interference and provide a better guarantee to the service quality of microservices. However, stateful services including database

[1]The trace is now open for public access as part of the Alibaba Cluster Trace Program via https://github.com/alibaba/clusterdata.

and Memcached are deployed in a dedicated cluster which is not shared with other batch applications or stateless services.

2.2.2 Microservices system metrics. Alibaba makes use of the Application Real-Time Monitoring Service system to collect microservice traces [2], which is similar to Dapper [26]. As shown in Fig. 2(b), the microservice monitoring system collects several system metrics for each container produced in every minute and takes the average to record. These metrics range from hardware-layer such as cache misses per kilo instructions and cycles per instruction to operating-system, including CPU utilization and memory utilization, and also contains application-layer index such as Java virtual machine (JVM) heap utilization and JVM garbage collection (GC). *Values* denote the exact number of these metrics, and *Timestamp* is the time when a corresponding metric is collected. *Pod IP* is the IP address of the pod in where a *Microservice* is deployed. A microservice usually runs in hundreds of containers.

2.2.3 Microservice invocations in a call graph. The microservice monitoring system also records in detail the call dependency between related microservices within a call graph as shown in Fig. 2(c). All invocations (or calls) between microservices triggered by the same user request share one unique *TraceID*, which is the identifier of a call graph. Moreover, an upstream microservice (*UM*) shall call a downstream microservice (*DM*) via a specific *interface*. The IP addresses of the pods holding them are recorded by the system as well, i.e., *UM Pod_IP* and *DM Pod_IP*. The trace also contains the response time (*RT*) of each call. Moreover, each call is identified by a unique *rpcID*, which contains the ID information of a pair of microservices. For example, *rpcID 0.1.1* and *0.1.2* denote two calls sent from the same UM to two different DMs. To avoid running out of storage space, the monitoring system only records a certain number of call graphs which are sampled based on *TraceID*. For all calls in each *TraceID*, the system also records their communication paradigm. Among all the calls that happened between two stateless microservices in the traces, RPC, MQ and IP account for 76%, 23% and 1% of communication paradigms respectively. As such, MQ contributes to a non-negligible percentage of calls in production clusters, which is quite different from the synthetic benchmarks provided by the academic, since the latter rarely adopts MQ as a communication paradigm.

2.2.4 Aggregate invocations. From a single microservice point of view, the monitoring system also records all the calls (received from UMs or sent to DMs) related to each individual microservice. As shown in Fig. 2(d), a *microservice* contains multiple *provided interfaces* to be called by its *UMs*. It calls *DMs* via different *consumed interfaces*. Correspondingly, *call times* quantifies the number of calls generated from each interface in one minute with the time recorded by *TimeStamp*. It is worth noting that, these calls are produced by all call graphs containing the target microservice and therefore, the traces only present the aggregate results. In addition, *RT* characterizes the average response time among all these calls within one minute for each interface.

3 ANATOMY OF CALL GRAPHS

In this section, we present a comprehensive study about the graph topology of microservice call graphs. We first show these call graphs present several distinct features and are substantially different from traditional DAG graphs of batching processing jobs (§ 3.1). We then show how to apply graph learning algorithms to cluster the call graphs of each online service into multiple classes (§ 3.2). Last, we dissect microservice call graphs to focus on smaller components in each tier for generating new microservice benchmarks at a large scale (§ 3.3).

3.1 Characteristics of microservice call graphs

To explore how microservices differ from traditional batch applications, we quantify the statistics of microservice call graphs in terms of the number of microservices, the call depth, and the in-degree (out-degree) of each microservice.

The size of microservice call graphs follows a heavy-tail distribution. While most call graphs contain a small number of microservices and have three tiers, a non-negligible number of graphs are big and deep. As shown in Fig. 3(a), the number of microservices in a graph follows a Burr distribution [19]. In particular, more than 10% of call graphs contain more than 40 unique microservices. The largest call graph can even consist of hundreds to thousands of microservices. This result indicates that the scale of existing benchmarks is far smaller than that in real traces [16, 27, 43]. For these call graphs of large size (containing more than 40 microservices), about 50% of their microservices are Memcacheds. We observe that this percentage is 20% higher than that under those call graphs of small size. Since getting (hot) data from Memcacheds is much faster than from databases, maintaining a large number of Memcacheds can significantly reduce the RT of complicated services.

We proceed to study the graph depth of each call graph. Interestingly, as depicted in Fig. 3(b), a common graph depth in Alibaba traces is three. The reason behind this is that, when serving an online request in Alibaba cluster, a microservice usually calls multiple downstream microservices, which will then query data from MCs directly as such data is usually hot data that is frequently accessed by other requests and cached in Memcacheds, e.g., the information of goods in an online store. Quantitatively, the call graphs have an average

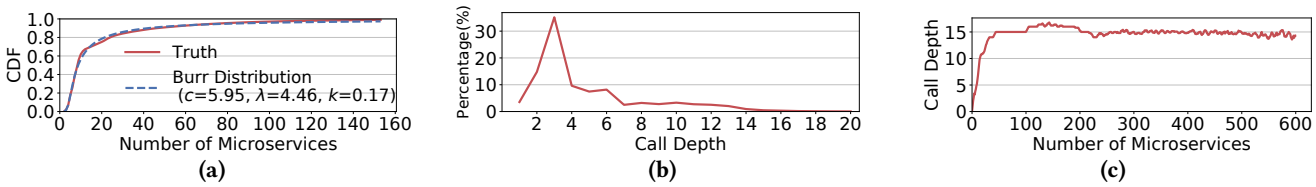


Figure 3: Statistics of all microservice call graphs in Alibaba traces. (a) Cumulative distribution of the number of microservices in each call graph. For ease of presentation, we cut off the long tail to only count those numbers that are within the 99th percentile. (b) The distribution of call depth in all call graphs. (c) The maximum call depth (95th percentile) under a fixed number of microservices.

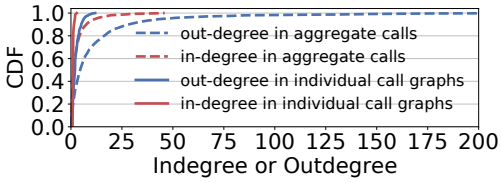


Figure 4: Distribution of the degree of stateless microservices in individual graphs and aggregate calls.

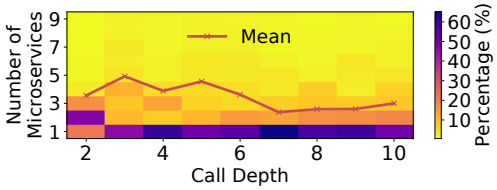


Figure 5: The distributions of the number of microservices in different tiers.

depth of 4.27, with a stand derivation of 3.25. As such, the call depth of microservice graph is in general shorter than the critical path length presented by DAG graphs from batch applications in Alibaba clusters, which is reported in [31]. Nevertheless, more than 4% of call graphs present a call depth of more than ten. In such cases, it is extremely challenging to configure the right number of containers for all microservices in production clusters, using conventional deep-learning based approaches, like those in [17, 40]. These approaches encode all the associated microservices of different tiers in a graph with timely resource allocations as the input vector to a neural network. As a result, they can easily yield a large model size and lead to overfitting since the number of samples with each one having both a large graph size and a negative label (e.g., RT violation) is small. It is therefore desirable to find an alternative approach that can efficiently allocate resources for microservices in a large call graph.

Microservice call graph behaves likes a tree and many of them only contain a long chain. As shown in Fig. 3(c), the call depth stagnates when the number of microservices increases. This is due to that a microservice graph tends to branch out quickly like a tree to include more two-tier invocations. Once a call is sent to a stateful microservice, it will not incur further calls. Again, this scattering property is different from that observed from traditional DAG graphs, which usually contain both scatter and gather components. To further validate this argument, we present the distribution of both in-degree and out-degree of microservices in call graphs and show the result in Fig. 4. More than 10% of stateless microservices have an out-degree of at least 5, while most microservices have an in-degree of one (i.e., only one UM in a call graph). As a comparison, more than 99% of vertices in DAG graphs have out-degrees no more than 3 while their in-degrees follow a long-tail distribution. When we examined the distribution of the number of microservices in each tier, we found that many tiers have only one microservice. As shown in Fig. 5, as long as the depth becomes larger than two, the corresponding tier includes only one microservice with a high probability (above 60%). As such, many deep graphs can be represented by one long chain. For these graphs, detecting the bottlenecked microservice is relatively easy. One can efficiently derive the processing time of each individual microservice along the chain and check whether an overload occurs based on information from historical traces.

Many stateless microservices are hot-spots. To quantify to what extent a single microservice can be shared by all call graphs, we explore the distribution of in-degree (out-degree) of stateless microservices in aggregate calls. Aggregate calls count all the invocations related to each individual microservice from all call graphs. As depicted in Fig. 4, more than 5% of microservices have in-degrees of 16 in aggregate calls. These super microservices appear in nearly 90% of call graphs and handle 95% of total invocations in Alibaba traces. This result implies that, the loosely-coupled microservice architecture leads to a significant unbalance of workload across different microservices. This is beneficial for resource scaling since the system manager shall only focus on the scaling of individual microservices and allocate much more containers to these super microservices.

Microservice call graphs are highly dynamic. Another distinction of microservice call graphs is that they are dynamic, in other words, they can present significant topological differences between each other even among all the graphs generated by the same online service. Each online service is represented by an entering microservice which is called by a user directly (e.g., Microservice A in Fig. 1(a)) and there

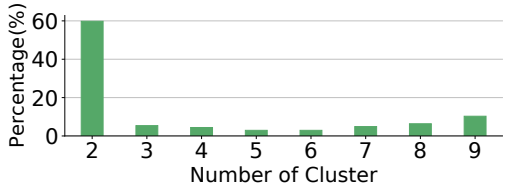


Figure 6: The distribution of the number of classes of graph topologies in all online services.

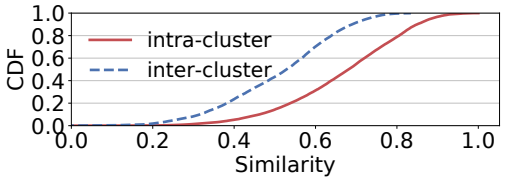


Figure 7: Intra-cluster similarity v.s. inter-cluster similarity.

are more than 3000 different services in total in Alibaba traces. Once a call is sent to an entering microservice, the subsequent calls can be quite complicated depending on the status of a user (e.g., whether a user holds a coupon ticket when making an online ordering). We apply graph learning algorithms to cluster microservice call graphs into different clusters based on their topology (§ 3.2). As shown in Fig. 6, all online services have at least two classes of graph topologies. In particular, more than 10% of services are implemented in nine clusters. This further imposes a great challenge on graph-based prediction tasks on microservices. The existing CNN-based approach for microservice resource management fails to characterize these dynamics and is not applicable to real-life industry applications, (e.g., [17, 40]).

3.2 Graph learning algorithms

In this part, we present the graph learning algorithms that are used to classify the call graphs of each microservice into different classes. Our learning algorithms can differentiate graphs based on their topology as well as the composition of microservices. The key step is to encode each microservice call graph into a vector. To achieve this, we adopt the recently developed graph learning scheme, i.e., *InfoGraph* [29]. *InfoGraph* is an unsupervised approach that takes both the node information (e.g., the type of a microservice) and the edge attribute (i.e., the communication paradigm) along with the adjacent matrix as an input to the deep neural network. By maximizing the mutual information conveyed in a training set, *InfoGraph* manages to generate an embedded vector for each graph. We train each online service separately and apply K-means clustering on the embedded 20-dimensional vectors to group all the call graphs generated by this service into multiple classes. The number of classes is set to the candidate in the range [2,10] that can yield the highest Silhouette score.

For examining the goodness of these clustering results, we further apply a widely-adopted method, i.e., *Graph Kernel* to compute the similarity between any two graphs of a service. *Graph Kernel* defines a kernel that captures the semantics inherent in the graph structure and is reasonably efficient to evaluate [36]. We quantify both the intra-cluster and the inter-cluster similarity for each service. In the former, we compute the similarity between any two graphs within each cluster and take the weighted average whereas in the latter, we average the similarities between any two graphs generated by a service. The whole clustering process is described in Algorithm 1. In Step 3, we randomly sample m graphs in each cluster for better scalability since the *Graph Kernel* method has a complexity of $O(n^4)$ where n is the number of nodes. We quantify the similarity between the clustered graphs in Fig. 7 ($m = 50$) and it shows that the intra-cluster similarity is 30% higher (in average) than the inter-cluster similarity, suggesting the designed clustering algorithm is quite effective to distinguish between different call graphs.

Algorithm 1: Clustering microservice call graphs

input : G: a set of Graphs for a service, l: embedding length, m: sample number
output: Clustering result C, Graph similarity S

1 $G_{embedding}$ = InfoGraph(G, l) ;
2 C = Kmeans($G_{embedding}$) ;
3 G_{sample} = Sample(C, m) // sample m graphs randomly in each cluster
4 $S_{interCluster}$, $S_{intraCluster}$ = S(G_{sample})

3.3 Anatomical analysis

To inspect how a call graph is formed and guide the generation of new microservice benchmarks, we study in this section the detailed structure of two-tier invocations in different tiers among all call graphs.

The call patterns of stateless microservices vary a lot over different tiers. In the traces, a noticeable fraction of stateless microservices are *blackholes* since they have no DM. Whenever a call is sent to a blackhole, the call graph will stop to branch out. On the contrary, there exists another type of *relays* microservices that will inevitably call others to serve the user requests. The rest of microservices (or *normals*) shall call their DMs with a certain probability. Moreover, the distribution of these three types varies over different tiers. As shown in Fig. 8(a), the percentage of *black holes* (*relays*) increases (decreases) with the call depth growing, which is in line with the observation that the expected call depth in a call graph is short. To make things even more complicated, the probability that whether a *normal* microservice will call other microservices is still tier specific. Similar to the patterns of *relays*, one may also expect the probability that a

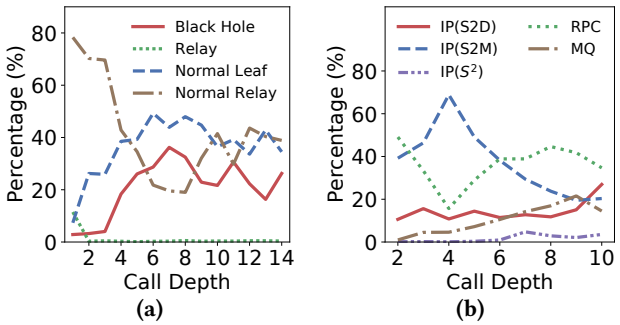


Figure 8: Tier-level microservice statistics. (a) The percentage of different types of stateless microservices changes over tiers. (b) The distributions of communication paradigms in different tiers.

normal proceeds to call DMs (i.e., the percentage of *normal relays* in Fig. 8(a)) decreases over tiers. Contrary to this expectation, when the call depth is above 8, such a probability increases over tiers. In a conclusion, microservice graphs have a lot of distinct features and it is quite challenging to simulate production call graphs using simple mathematical models.

MQ contributes greatly to reducing the end-to-end RT in deep graphs. In addition to the number of microservices, the communication paradigm also varies significantly over tiers. As depicted in Fig. 8(b), the percentage of communications between stateless microservices and Memcacheds (i.e., S2M) reduces linearly in call depth when the depth is above three. It indicates the cache miss rate of queries increases quickly when call graphs become deeper and deeper. When the data misses a hit in caches, this query will be sent to database service. This is in line with the result illustrated in Fig. 8(b). The percentage of communications between stateless microservices and databases (i.e., S2D) increases sublinearly when the depth increases. The gap between the change rates of these two communication paradigms is filled by MQ, whose percentage also increases in call depth. Since MQ is an indirect one-way communication under which the call can be handled in the backend without immediate reply, a large percentage of MQs can help to greatly reduce the end-to-end RT when the call graph is deep.

4 DEPENDENCY BETWEEN STATELESS MICROSERVICES

Light-weight microservices form complicated and dynamic call dependencies. These are fundamentally different from DAG dependencies [12, 15, 38]. In this section, we explore the call dependencies between stateless microservices to develop understandings of how to optimize microservice designs, so as to reduce the communication overhead and avoid possible deadlocks.

Table 1: Cyclic dependency between a pair of microservices via different communication paradigms.

UM \ DM	RPC	MQ
RPC	7.6%	0.0016%
MQ	0.00167%	0.22%

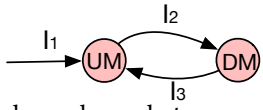


Figure 9: Cyclic dependency between a pair of microservices.

4.1 Cyclic dependency

In this part, we show how easily a cyclic dependency can occur in Alibaba microservice traces. An example of cyclic dependency is illustrated in Fig. 9 where a DM replies to its UM immediately without involving other microservices. Furthermore, cyclic dependency can be categorized into two classes, i.e., strong dependency and weak dependency [22]. In the strong case, the entering interface of UM is the same as the reply interface for DM calls to call (i.e., $I_1 = I_3$). By contrast, these two interfaces are different in weak dependency (i.e., $I_1 \neq I_3$). Strong cyclic dependency can lead to a deadlock if not designed carefully and would be better off shipping the cyclic bits together as a larger service [18].

Cyclic dependency makes up a non-negligible fraction among all dependencies. As quantified in Table 1, cyclic dependency contributes to more than 7.8% of the total microservice dependencies and most are via RPC calls. Among all these cyclic dependencies, 2.7% of them are strong dependencies. Moreover, for each pair of microservices that have a strong dependency, whenever an interface of a UM calls a DM, the DM would call back the UM subsequently via the same interface with a probability as high as 83%. We also investigate the number of cyclic calls involving three microservices and this number is relatively small, i.e., less than 200 among all the traces with billions of calls. As such, long cyclic calls rarely happen. As a consequence, the service provider should only pay much attention to those pairs of two microservices that form a strong dependency and validate whether there is a strong need to combine the two interfaces into a single one (e.g., I_1 and I_2) so as to avoid deadlocks.

4.2 Coupled dependency: dependency with high call probability and large call time

Prior online service systems assume an upstream service calls its downstream services following a fixed probability distribution, e.g., [20], [34]. However, this assumption may not hold in practice in microservice systems since a UM can repeatedly call the same DM multiple times in a two-tier invocation. To model such a dependency, we adopt call time

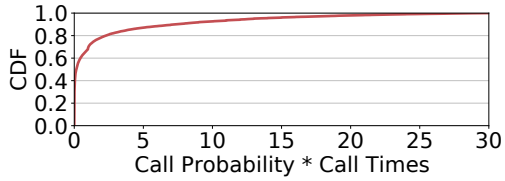


Figure 10: Cumulative distribution of the product of Call Probability and Call Time between all pairs of microservices. A high product means the DM will be called by UM within the same pair repeatedly with a high probability.

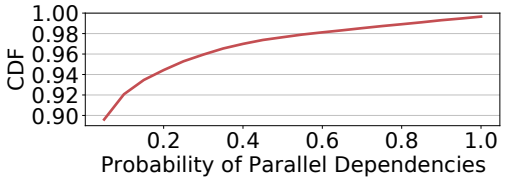


Figure 11: Cumulative distribution of the probability of parallel dependency between all pairs of microservices. A parallel dependency with high probability means a pair of microservices will be called by others in parallel in most times.

as a complementary metric to call probability as follows:

$$\text{Call Probability}(Y2X) = \text{Count}(X) \;/\; \text{Sum}. \quad (1)$$

$$\text{Call Time}(Y2X) = \text{Count}(X) \;/\; N. \quad (2)$$

where Count(X) is the number of times that UM Y calls DM X (note that a DM may be called multiple times by Y within the same two-tier invocation), and Sum denotes the number of two-tier invocations trigger by Y in all call graphs. In Eq. (2), N is the number of those two-tier invocations in which DM X was called.

Call probability measures for each pair of microservices how likely the DM is called while call time quantifies how many times the DM is called in each two-tier invocation initiated by the UM. When both of Call Probability($Y2X$) and Call Time($Y2X$) are large and beyond certain thresholds, i.e., 2 and 0.9 respectively, Y and X and their corresponding interfaces then form a strong coupled dependency.

A noticeable fraction of pairs have strong coupled dependency and their interfaces could be coupled together for performance optimization. We show the distribution of the product of Call Time and Call Probability in Fig. 10. Surprisingly, more than 10% of pairs of microservices have a product of no less than five, implying that a lot of microservice pairs in Alibaba cluster have a strong coupled dependency. We continue to quantify the percentage of these pairs whose DM are not called by other microservices (i.e., the DM has an in-degree of one). The result indicates that 17% of pairs with strong coupled dependency do not share DM with any other microservice. For these pairs, coupling the called interface of DM with the corresponding interface of UM together can substantially reduce the communication overhead since coupling can replace a remote call with a local self-call to reduce network traffic.

4.3 Parallel dependency

In a two-tier invocation, a UM calls its multiple DMs either in a sequential manner or in parallel. Parallel dependency can help to greatly reduce the RT of upstream microservices. For ease of analysis, in this part, we only examine parallel dependency between each pair of two microservices in Alibaba traces.

Strong Parallel Dependency rarely exists in Alibaba traces. The probability of a parallel dependency measures how often a pair of microservices shall be called by a UM in parallel among all the two-tier invocations involving this pair. We measure the probability of parallel dependency for all pairs of microservices. Fig. 11 depicts the CDF of these probabilities and it shows that, 10% of pairs of microservices have a parallel dependency with probability larger than 0.05. However, only 0.6% of pairs show a strong parallel dependency (with a probability larger than 0.9). To sum up, there is a small number of strong parallel dependencies in Alibaba traces. However, in case of such dependency, it is suggested to couple the two called interfaces into a single microservice such that, a UM only needs to call one microservice instead of calling two interfaces in parallel from different microservices. By doing so, the communication overhead can also be reduced.

5 MICROSERVICE RUNTIME PERFORMANCE

Understanding the runtime performance of microservices is critical to ensure the quality of services. Since microservices run in hundreds to thousand of containers in a hybrid cluster and serve time-varying requests with highly dynamic call dependencies, several factors can affect the RT performance of microservices. In this section, we study the impact of graph topology and resource interference together with microservice call rates (MCR) on the RT performance. We also characterize the relationship between MCR and several OS-level and application-level metrics to quantify the resource pressure in Alibaba clusters.

5.1 Microservice call rate

Microservice call rate (MCR) measures the number of calls received by a microservice in each minute per container. It is expected that the resource utilization of a running container highly relates to MCR and therefore, a large MCR will lead to a high resource pressure. To examine this, we adopt Spearman Correlation as an evaluation metric, which measures between MCR sequences and the container running metric

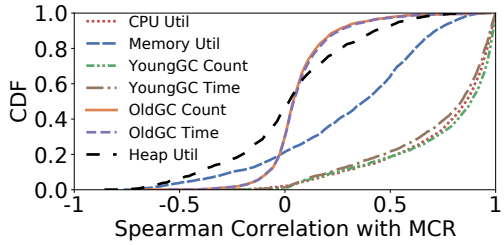


Figure 12: The correlation between microservice call rate and different performance metrics.

sequences for each microservice. In Fig. 12, we plot the distribution of Spearman Correlations between MCR and multiple different metrics in both OS-level and application-level.

Microservice call rates highly correlate with CPU utilization and Young GC but not with memory utilization. As illustrate in Fig. 12, all microservices show a positive correlation between the CPU utilization and MCR and more than 80% of them yield a strong correlation (with Spearman Correlation bigger than 0.6). YongGC Count and YoungGC Time also show a strong correlation with MCR. By contrast, more than 20% of microservices have a negative correlation between MCR and memory utilization. This implies that CPU utilization and Young GCs are much better indicators to reflect resource pressures of running containers of a microservice comparing to memory utilization. A key reason behind is that, the memory utilization is almost stable at runtime in most containers in Alibaba microservice traces (with a variance of less than 10%). Our finding also matches the observation found in [21], i.e., most containers exhibit steady memory but their CPU utilization varies in production clusters.

5.2 Microservice RT performance

In this part, we investigate how the microservice response time (RT) can be impacted by other factors including the complexity of call graph, resource interference, and MCR, and to what extent these impacts can be.

End-to-End RTs of an online service are stable among call graphs of similar topologies but vary significantly across different topologies. We apply Algorithm 1 to cluster all the call graphs of each service into multiple classes. As stated in § 3.2, the features selected by *InfoGraph* include the graph topology along with the types of microservices in a graph. As such, each class contains graphs of similar topology and call paths. Within each class, we compute both the standard derivation and the mean of the end-to-end RTs (i.e., RTs of the *Entering Microservice*) and then take the ratio between them as a measurement of the intra-cluster-variance. Similarly, we collect all end-to-end RTs from all classes of a service to measure the inter-cluster-variance. We plot in Fig. 13 the cumulative distribution of intra-cluster-variance to inter-cluster-variance ratios. The results show that more

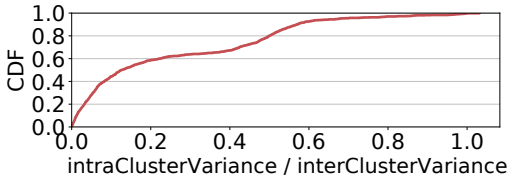


Figure 13: Cumulative distribution of RT intra-cluster variance to RT inter-cluster variance ratios.

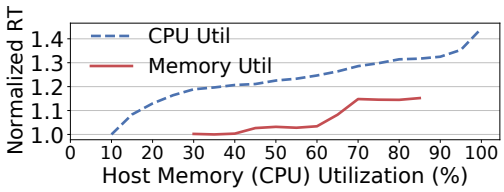


Figure 14: Performance degradation due to resource interferences on the same physical host. X-axis represents the host CPU (memory) utilization while Y-axis quantifies the 75th percentile normalized RT among all microservices.

than 90% of online services have a small ratio (less than 0.6), indicating the RTs within each cluster are much more stable than that across different clusters. This further implies the graph topology has a heavy impact on the end-to-end RT and moreover, our designed graph learning algorithm can be applied to predict the RT performance.

RT performance can be greatly degraded due to a high host CPU utilization. In Alibaba clusters, online microservices usually co-exist with batch processing jobs on the same physical host to improve cluster utilization. As such, resource interference can easily occur. In this part, we continue to evaluate how seriously resource interference can impact the response of a microservice. Indeed, requests to a microservice deployed in a host can further call multiple downstream microservices (DM) which are deployed in different hosts. However, these requests are evenly distributed to all the instances of each DM. In this sense, the impact of downstream hosts can be treated as a constant when the utilization of this host changes. To cancel the effect caused by different microservice call rates, we collect for each fixed call rate (MCR) all the RTs under different host utilization. We then take the RT under a low host utilization (i.e., 10%) as a baseline and measure the normalized RTs under different host utilization for a fixed MCR. We then average all the normalized values across different MCRs under each host utilization. Finally, we compute the 75th percentile normalized RT among all microservices. Fig. 14 depicts the normalized RTs under different host CPU and memory utilization. One can observe that the host CPU utilization has a heavy impact on the RT performance. When the host CPU utilization exceeds 40% (80%), the RT of a microservice can be degraded by more

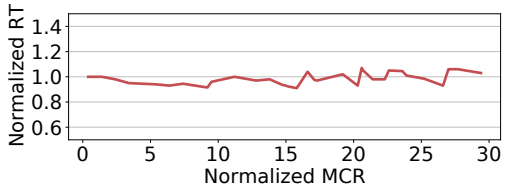


Figure 15: RT performance under different normalized microservice call rates.

than 20% (30%) in average. However, the host memory utilization has a much lighter impact on the RT. When the host memory utilization is below 60%, the interference can be ignored. These results indicate most online microservices are sensitive to CPU interference and there is a strong demand for a more efficient resource scheduler that can well balance the CPU utilization across different hosts. We observe from traces that the variance of CPU utilization across hosts in each minute can be as high as 20%, implying there is a large room to balance the batch workload across hosts.

RTs of a microservice are stable when the call rate varies. From a modeling perspective, the response time of a system is a function of the request arrival rate and the amount of allocated resources. Since Alibaba cluster applies the same configuration to all the running containers deployed for each microservice and perform load balance among all containers, the resulted RT should only depend on the microservice call rate per container (MCR), when fixing the type of graph topology and the host resource utilization. To investigate such an effect, we characterize for each microservice the impact of MCR on RT performance by averaging all the RTs under each normalized MCR. We take the 75th percentile RT among all microservices and plot the result in Fig. 15. It shows that RTs of most microservices are stable when the call rate varies. This is due to most calls in Alibaba clusters can be processed immediately without any queueing delay. Even for a large MCR (95th percentile), the CPU utilization of most running containers within a minute is below 10%. These results also indicate there is a large room to improve the resource utilization of microservices by resizing a proper number of running containers.

6 PROBABILISTIC MODEL FOR MICROSERVICE GRAPH GENERATION

Having figured out several statistical properties of microservice call graphs, in this section, we investigate a probabilistic model to mathematically formulate the simulation of new microservice graphs that can well preserve these observed properties from Alibaba traces. We believe such a simulation is beneficial for the generation of other large-scale microservice benchmarks that can be used for research on microservice resource scheduling problems, e.g., [17, 23, 40].

6.1 Stochastic graph modeling

As investigated in § 3.1, a microservice call graph is scattered to include multiple two-tier invocations. Following this observation, a graph $G_s = (V_s, E_s)$ is initialized by a starting service s and can grow with more and more two-tier invocations involved. Each microservice $v \in V_s$ has a tier number (call depth) $d(v)$, e.g., the tier number of s is $d(s) = 1$. The largest tier number in G_s is denoted by h_s. Each edge $e(v_i, v_j)_{i<j} \in E_s$ is directed and formed by one parent v_i and one child v_j. In addition, each $v \in V_s$ has a label $l(v)$ that denotes its type of service, i.e., $l(v) \in \mathbf{L} = \{database, Memcached, blackhole, relay, normal\}$. Since most edges in Alibaba traces only span two adjacent tiers, we consider $d(v_j) = d(v_i) + 1$ whenever $e(v_i, v_j) \in E_s$.

Let $C(v) = \{v_c : (v, v_c) \in E_s\}$ denote the children set of v. For $v \in V_s$ with $d(v) = h$, $|C(v)|$ follows a random distribution given by:

$$\Pr\big(|C(v)| = j\big) = F_h(j), \tag{3}$$

where F_h is the distribution of the number of microservices in a two-tier invocation starting from tier h in Alibaba traces. Once node v branches out to contain j children, i.e., $|C(v)| = j$, all the children in $|C(v)|$ share the same tier number of $h+1$. Furthermore, each child $v_c \in C(v)$ takes a label from $\mathbf{L}$ randomly based on the following distribution:

$$\Pr\big(l(v_c) = \phi\big) = G_{h+1}(\phi), \quad \forall \phi \in \mathbf{L}, \tag{4}$$

where $G_{h+1}(\phi)$ can be simply derived by combining results from Fig. 8(a) and Fig. 8(b). More specifically, when ϕ = DB (MC), $G_{h+1}(\phi)$ is the percentage of S2D (S2M) communications in tier $h+1$ in Fig. 8(b). By contrast, when ϕ = *blackhole*, $G_{h+1}(\phi)$ is the percentage of RPC communications (quantified in Fig. 8(b) multiplied by the percentage of *blackhole* Fig. 8(a) in tier $h+1$. Similar results hold when ϕ = *relay* or *normal*. v_c continues to branch out following the same procedure when its label $l(v_c)$ is a *relay*. In addition, when $l(v_c)$ = *normal*, v_c also has a chance to branch out and the probability is the same as the percentage of *normal relays* in each layer as illustrated in Fig. 8(a).

6.1.1 Graph refinement. In the model described above, each child can only keep one parent. Indeed, the result uncovered in § 3.1 suggests that most DMs only have one UM (i.e., with in-degrees of one). However, there still exists a lot of exceptions under which the in-degree of a DM is much larger than one. To take into account this fact, we continue to make refinement of the graph model. More specifically, we adopt the result that quantifies the number of microservices distributed in each tier to allow some nodes to have more than one parent.

Let N_h be the original set of nodes generated in G_s in tier h before refinement and R_h be the number of nodes after refinement. Further denote by P_{h-1} the set that contains

nodes in tier $h-1$ with each connected to at least one child, we have $N_h = \bigcup_{p \in P_{h-1}} C(p)$ and

$$\sum_{p \in P_{h-1}} \big|C(p)\big| = \big|N_h\big|. \tag{5}$$

R_h shall be sampled from a random distribution, i.e.,

$$\Pr\big(R_h = m\big) = \Phi_h(m), \tag{6}$$

where Φ_h is the distribution of the number of services in tier h (illustrated in Fig. 5). The refinement is necessary only when $R_h < |N_h|$. In this case, refinement is conducted by merging $R_h - |N_h|$ node pairs into $R_h - |N_h|$ individual nodes in tier h. It is worth noting that, all the nodes in each node pair share the same label and their parents are different with each other before merging. We choose these node pairs in a sequential manner. Each time, we randomly select two parents and merge two children of them if they share the same label. The merged node will then connects to two parents.

6.1.2 Generator of call graphs calls. Based on the built stochastic graph model, we implement a graph generation algorithm in this section to synthesize large-scale microservice traces. To mimic the call dependency, our generator relies on combining several two-tier invocations together in a sequential manner to add more and more calls, i.e., Step 15 in Algorithm 2. Each call has a communication paradigm (i.e., *comPara* in Step 12) and a unique identifier represented by a *rpcId* (i.e., Step 13). *rpcId* can help to capture the dependency of all pairs of microservices in the whole graph. Each two-tier invocation is generated in three steps. Firstly, the generator will check whether a stateless service *UM* with a generated call depth based on a *rpcId* (Step 8) is a *relay* or *normal relay* in Step 9. Secondly, the generator will yield a random number of DMs following a distribution in the corresponding tier in Step 10. Thirdly, the generator will determine the *comPara* of each pair of microservices in Step 12. These three steps shall iterate and be terminated when the queue that stores all stateless services (in Step 17) is empty. Finally, the generator will perform the graph refinement, i.e., Step 18 to Step 24.

Algorithm 2: Call Graph Generator

```
1  L Call Graph List: [rpcId, UM, DM, ComPara];
2  Q Queue for Stateless Services;
3  L.add(0, User, EnterMicroservices, Http);
4  Q.push([0, EnterMicroservices]);
5  /* Graph Generator                                  */
6  while !Q.isEmpty() do
7      [rpcId, UM] = Q.pop();
8      depth = Depth(rpcId) + 1;
9      if Relay(depth) then
10         num = Node(depth);
11         for i ∈ num do
12             compara = ComPara(depth);
13             rpcIdChild = RpcIdExtend(rpcId, i);
14             DM = RandomName(rpcIdChild);
15             L.add(rpcIdChild, UM, DM, comPara);
16             if DM is stateless then
17                 Q.push([rpcIdChild, DM]);
18 /* Graph Refinement                                 */
19 for i ∈ callDepth − 1 do
20     j = i + 1;
21     P_j = Pair(L, j);
22     for p ∈ P_j do
23         if Refine(p) then
24             DM_u = Rename(p^1_DM, p^2_DM)
                update(L, DM_u, p^1_DM, p^2_DM)
```

6.2 Theoretical analysis

To validate whether the graph model in § 6.1 can well preserve these graph properties explored from Alibaba traces, we turn to the theoretical characterization of its stochastic properties. Since the graph modeling is based on the distribution of microservice degrees, we focus on quantifying the other two distributions, i.e., the distribution of graph depth and the number of nodes in a graph. Our major results are illustrated in the following theorems.

Theorem 6.1. *In the graph model, let $\Gamma(k) = Pr(h_s = k)$ be the probability that a randomly generated graph G_s has exactly k tiers, when $k \geq 2$, $\Gamma(k)$ is expressed by:*

$$\Gamma(k) = \left(1 - \sum_{h=1}^{k-1} \Gamma(h)\right) \cdot \sum_{n=1}^{U} \Phi_k(n)\big(1 - G_k(\boldsymbol{r}) - G_k(\boldsymbol{nr})\big)^n, \tag{7}$$

where $G_k(\boldsymbol{r})$ ($G_k(\boldsymbol{nr})$) given by Eq.(4) quantifies the probability that a microservice is a relay (normal relay) *in tier k. And $\Phi_k(n)$ given by Eq.(6) denotes the probability that there are exactly n microservices in tier k. U is an upper bound for the number of microservices in each tier.*

Proof. We prove this theorem by mathematical induction.

When $k = 2$, i.e., the graph only has two tiers, it must be the case that all the microservices in Tier 2 are not *relays*. In addition, when a microservice is a *normal*, it can not branch out. By conditioning on the number of microservices in Tier 2, we have:

$$\begin{aligned} \Gamma(2) &= \sum_{n=1}^{U} Pr(R_2 = n) \cdot Pr(h_s = 2 \big| R_2 = n) \\ &\overset{(a)}{=} \sum_{n=1}^{U} \Phi_2(n)\big(1 - G_2(\mathbf{r}) - G_2(\mathbf{nr})\big)^n, \end{aligned} \tag{8}$$

where (a) is due to the probability that each microservice cannot branch out is equal to $\big(1 - G_2(\mathbf{r}) - G_2(\mathbf{nr})\big)$. As such, Theorem 6.1 holds for $k = 2$.

Assume this result holds for all $k \leq \rho - 1$. When $k = \rho$, let $\mathbf{E}_\rho$ denote the event that all nodes in tier ρ do not branch out, we have:

$$\Gamma(\rho) = Pr(h_s \geq \rho) \cdot Pr(\mathbf{E}_\rho | h_s \geq \rho). \tag{9}$$

Similar to the argument as that in Eq. (8), the second term in the R.H.S. of Eq.(9) is given by:

$$Pr(\mathbf{E}_\rho | h_s \geq \rho) = \sum_{n=1}^{U} \Phi_\rho(n) \big(1 - G_\rho(\mathbf{r}) - G_\rho(\mathbf{nr})\big)^n. \tag{10}$$

Combining Eq.(9) and Eq.(10) together immediately implies the result, this completes the proof of Theorem 6.1. □

Theorem 6.2. *Let $\Psi(n) = Pr(|V_s| = n)$ be the probability that a randomly generated graph G_s has exactly n nodes, $\Psi(n)$ is given by:*

$$\Psi(n) = \sum_{k=1}^{D} \Gamma(k) \cdot \sum_{\sum_{i=1}^{k} n_i = n} \prod_{i=1}^{k} \Phi_i(n_i), \tag{11}$$

where $\Phi_i(n_i)$ is the probability that there are n_i microservices in tier i and D is the maximum call depth in a graph.

Proof sketch. By conditioning on the number of tiers that G_s has and enumerating all possible combinations of the node distributions in each tier, Theorem 6.2 follows.

6.3 Evaluation

In this section, we evaluate the effectiveness of our graph generator algorithm as well as the correctness of the theoretical analysis. In particular, we quantify the distribution of call depth as well as the number of microservices in the generated graphs, and compare them to the numerical results derived based on the theoretical analysis and true distributions in Alibaba traces.

As shown in Fig. 16, in terms of the distribution of call depth, the generated call graphs match quite well with the theoretical analysis and almost preserve the ranking between the probabilities of all depths in Alibaba traces (with a KL-divergence of 0.16). One mismatching is that, the probability of having a depth of two in the generated graphs is larger than that in traces. A major reason is that, Alibaba traces are incomplete, e.g, a few calls are lost due to missing of microservice IDs. Fig.17 depicts the distribution of the number of microservices. It shows that the generated graphs are close to real call graphs in terms of this metric (with a KL-divergence of 0.05). Since we set an upper bound for the number of microservices in each tier to 20 in the theoretical analysis (per Eq.(11)) for a better computation efficiency, the resulted numerical result yields a small probability when the

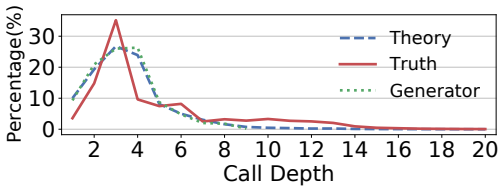


Figure 16: Distribution of call depths under the simulated microservice call graphs.

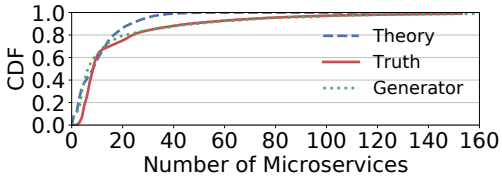


Figure 17: Distribution of the number of microservices under the simulated microservice call graphs.

microservice number exceeds 20. Nevertheless, it succeeds in capturing the long tail of true distributions (with a KL divergence of 0.06). These results demonstrate our built graph model is efficient to simulate microservice call graphs that can well preserve the major features of real traces.

7 RELATED WORK

Microservice benchmarks. Prior works use benchmarks, such as DeathStarBench [16], μSuite[27] and Acme Air [33] to study microservices in many different aspects. In particular, Gan *et al.* analyzed the difference between microservices and monolithic applications among networking, operating systems, cluster management, and programming frameworks in DeathStarBench [16]. Ueda *et al.* focused on the network overhead and explored how to reduce it under the microservice architecture [33]. Sriraman *et al.* found that the sub-ms-scale threading tends to be a new performance bottleneck for microservices under the three-tier microservice architecture since the latency of an RPC call is expected to be sub-milliseconds [28]. To tackle this issue, an automatic threading adaptation system was developed to enhance the performance of the RPC framework. However, all of these benchmarks are of a small scale and cannot reflect how microservices actually run in production environments.

Serverless benchmarks. Most recently, Yu *et al.* built a serverless benchmark to study the behavior of function as service [37]. This work shows that inter-function parallelization can lead to a better concurrency than in-function parallelization since the latter only runs the function in one instance with limited resource. As an implication, decoupling the parallelizable part in an application may help to speed up the process. In this paper, we provided insights on how

to couple interfaces from different microservices together so as to reduce the communication overhead.

Cloud workloads. There exist several public traces released from production clusters including Google Trace [24, 32], Alibaba Trace [21, 31], Azure Trace [14, 25], and so on. However, these traces contain batch jobs, online services, function as a service (FaaS), or even jobs running in a virtual machine. They are not for microservice architecture or lack of detailed information about microservices such as call dependencies between services. Recently, Zhou *et al.* designed DAGOR, a proactive load balancing system, to address the overload problem of microservices built for serving the WeChat system [42]. One fundamental limitation of this work is that, it does not conduct any analysis related to microservices in other aspects such as the call graphs. Shahrad *et al.* began to investigate the characteristics of FaaS workload of Azure Functions about their invocation frequencies [35], but they did not analyze the invocation dependencies between different functions.

Cloud trace analysis. In the literature, there exist a bunch of works that focus on the analysis of cloud workload, e.g., [11, 14, 21, 24]. However, they lack investigations on graph structures. Recently, Tian *et al.* conducted a comprehensive analysis of Alibaba DAG job traces [31]. While this work focuses on the characterization of task dependencies in DAG graphs as well as the trace synthesis, their observations are quite different from our findings on microservice call graphs. More importantly, we built a stochastic model to simulate call graphs based on a new classification of microservice types, which are fundamentally different from the graph synthesis in [31]. Moreover, we also conducted a solid analysis to validate the effectiveness of our model.

Performance characterization of online services. Existing works also adopt Markov Chain to simulate call dependencies between different online services [14, 20, 34]. However, Markov-chain based analysis does not apply to traces in a production cluster since microservices will repeatedly call the same downstream microservice multiple times with a specific order. To get away from complex call dependencies, many works adopt the queuing time in different layers, including operation system such as thread pools [42], software such as socket or hardware such as NIC [17], as the metrics to detect performance bottlenecks of microservices. However, the existing tracing systems cannot provide these metrics directly. In addition, the recently developed machine learning schemes are either based on CNN neural networks or reinforcement learning approaches, which cannot capture the dynamic dependency of microservice call graphs [17, 23, 40]. In this paper, we applied graph learning algorithms to represent the topology of graph dependency using an embedded vector, which can be used to analyze the RT performance of online services.

8 DISCUSSION AND CONCLUSION

In this paper, we conduct a comprehensive study of large-scale microservices deployed in Alibaba clusters. To the best of our knowledge, we are the first one to characterize in-depth the structural properties of microservice call graphs. Our study has revealed several important graph properties, i.e., microservice graphs are dynamic in runtime, most graphs are scattered to grow like a tree, and the size of call graphs follows a heavy-tail distribution. In this sense, existing open-sourced microservice benchmarks fail to preserve these properties since their scale is quite small and graph topology produced by an online service does not change much.

Our characterization of three types of call dependency between a pair of microservices shall provide insights on how to optimize microservice designs, such as balancing trade-offs between communication overheads and the ease of service development brought by the light-weight coupling.

Our study on microservice run-time performance provides several good implications on microservice scheduling and cluster resource management. In particular, the conclusion that most microservices are sensitive to CPU interference rather than memory interference can be leveraged to simplify the scheduling problem, i.e., it is significant to focus on the balance of CPU utilization across different hosts. In addition, the observation that RTs of online services highly depend on the call graph topology necessitates topology awareness for more efficient microservice schedulers, which however is quite challenging.

Our stochastic graph modeling makes a first attempt to simulate microservice call graphs in a large-scale. The conducted theoretical analysis provides a possible solution for quantifying properties of graphs with a well layered structure. One limitation, however, is that the graph model cannot handle the node sharing between different graphs, i.e., one node may appear in multiple graphs. This requires a further step towards matching nodes between all graphs based on the statistics collected from aggregated invocations. We shall leave this step as future work.

Another possible research direction is to apply our developed graph clustering algorithm as a starting point for implementing graph-based scheduling algorithms aiming at optimizing the end-to-end service response time.

9 ACKNOWLEDGEMENTS

This work is supported in part by Key-Area Research and Development Program of Guangdong Province under Grant No. 2020B010164003, the National Natural Science Foundation of China (No. 62072451), the Start-up Research Grant of University of Macau (SRG2021-00004-FST), and Alibaba Group through Alibaba Innovative Research Program.

REFERENCES

[1] 2021. Alibaba Cloud. https://www.alibabacloud.com/.
[2] 2021. Alibaba Cloud Product. https://www.alibabacloud.com/product.
[3] 2021. Amazon Web Services. https://aws.amazon.com/.
[4] 2021. CNCF. https://www.cncf.io/.
[5] 2021. Google Cloud. https://cloud.google.com/.
[6] 2021. Kata Containers. https://katacontainers.io/.
[7] 2021. Kubernetes. https://kubernetes.io/.
[8] 2021. Mysql. https://www.mysql.com/.
[9] 2021. Pods and Nodes. https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/.
[10] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *Proceedings of USENIX NSDI*. 419–434.
[11] George Amvrosiadis, Jun Woo Park, Gregory R. Ganger, Garth A. Gibson, Elisabeth Baseman, and Nathan DeBardeleben. 2018. On the diversity of cluster workloads and its impact on research results. In *Proceedings of USENIX ATC*.
[12] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).
[13] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of ACM SoCC*. 143–154.
[14] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of ACM SOSP*. 153–167.
[15] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.
[16] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of ASPLOS*. ACM, 3–18.
[17] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of ASPLOS*. 19–33.
[18] SE Ghirotti, T Reilly, and A Rentz. 2018. Tracking and Controlling Microservice Dependencies. In *Communications of the ACM*.
[19] Arief Hakim, I Fithriani, and Mila Novita. 2021. Properties of Burr distribution and its application to heavy-tailed survival time data. *Journal of Physics: Conference Series* (2021).
[20] Abhinav Kamra, Vishal Misra, and Erich M Nahum. 2004. Yaksha: A self-tuning controller for managing the performance of 3-tiered web sites. In *Proceedings of IWQoS*. IEEE, 47–56.
[21] Qixiao Liu and Zhibin Yu. 2018. The elasticity and plasticity in semi-containerized co-locating cloud workload: A view from Alibaba trace. In *Proceedings of ACM SoCC*. 347–360.
[22] Shangpin Ma, Chenyuan Fan, Yen Chuang I-Hsiu Liu, and Ciwei Lan. 2019. Graph-based and scenario-driven microservice analysis, retrieval, and testing. In *Future Generation Computer Systems*.
[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 *Proceedings of USENIX OSDI*.
[24] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of ACM SoCC*. 1–13.
[25] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *Proceedings of USENIX ATC*. 205–218.
[26] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. Dapper, a large-scale distributed systems tracing infrastructure. (2010).
[27] Akshitha Sriraman and Thomas F Wenisch. 2018. μ suite: a benchmark suite for microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 1–12.
[28] Akshitha Sriraman and Thomas F Wenisch. 2018. μTune: Auto-Tuned Threading for OLDI Microservices. In *Proceedings of USENIX OSDI*. 177–194.
[29] Fan-Yun Sun, Jordan Hoffman, Vikas Verma, and Jian Tang. 2020. InfoGraph: Unsupervised and Semi-supervised Graph-Level Representation Learning via Mutual Information Maximization. In *Proceedings of ICLR*.
[30] Andrew S Tanenbaum and Maarten Van Steen. 2007. *Distributed systems: principles and paradigms*. Prentice-Hall.
[31] Huangshi Tian, Yunchuan Zheng, and Wei Wang. 2019. Characterizing and Synthesizing Task Dependencies of Data-Parallel Jobs in Alibaba Cloud. In *Proceedings of ACM SoCC*. 139–151.
[32] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: the next generation. In *Proceedings of Eurosys*. 1–14.
[33] Takanori Ueda, Takuya Nakaike, and Moriyoshi Ohara. 2016. Workload characterization for microservices. In *2016 IEEE international symposium on workload characterization (IISWC)*. IEEE, 1–10.
[34] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. 2005. An Analytical Model for Multi-tier Internet Services and Its Applications. In *Proceedings of ACM Sigmetrics*. 291–302.
[35] Kaushik Veeraraghavan, Justin Meza, David Chou, Wonho Kim, Sonia Margulis, Scott Michelson, Rajesh Nishtala, Daniel Obenshain, Dmitri Perelman, and Yee Jiun Song. 2016. Kraken: leveraging live traffic tests to identify and resolve resource utilization bottlenecks in large scale web services. In *Proceedings of USENIX OSDI*. 635–651.
[36] S.V.N. Vishwanathan, Nicol N. Schraudolph, Risi Kondor, and Karsten M. Borgwardt. 2010. Graph Kernels. *Journal of Machine Learning Research* (2010).
[37] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing serverless platforms with serverlessbench. In *Proceedings of ACM SoCC*. 30–44.
[38] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceeding of USENIX NSDI*. 15–28.
[39] Xiantao Zhang, Xiao Zheng, Zhi Wang, Hang Yang, Yibin Shen, and Xin Long. 2020. High-density Multi-tenant Bare-metal Cloud. In *Proceedings of ASPLOS*. 483–495.
[40] 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 *Proceedings of ASPLOS*.
[41] Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. 2014. Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. In *Proceedings of VLDB*. 1393–1404.

[42] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. 2018. Overload Control for Scaling WeChat Microservices. In *Proceedings of ACM SoCC*. ACM, 149–161.

[43] 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 *Proceedings of ICSE*. IEEE, 323–324.