An In-Depth Study of Microservice Call Graph and Runtime Performance

Shutian Luo, Huanle Xu, *Member, IEEE*, Chengzhi Lu, Kejiang Ye, *Member, IEEE*, Guoyao Xu, Liping Zhang, Jian He, and Chengzhong Xu, *Fellow, IEEE*

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 also discover that the structure of call graphs for long-term developed applications is much simpler so as to provide better performance. Our investigation on microservice runtime performance indicates most microservices are much more sensitive to CPU interference than memory interference. Moreover, we design resource management policies to efficiently tune memory resources.

Index Terms—Trace analysis, microservice, performance characterization

1 Introduction

Loosely-coupled and light-weight microservices are gradually replacing monolithic applications [2]. Compared 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 [3], [4], [5].

To leverage microservice architecture, many benchmarks such as Acme Air [6], μSuite [7], and DeathStarBench [8] have been developed to explore the major characteristics of microservices. These benchmarks compare microservices and monolithic applications with respect to network overhead [6], remote procedure calls efficiency [7], and performance implications on resource management and hardware [8]. 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. 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 unique microservices, whereas most data-parallel jobs only contain a few stages.
- A microservice call graph has a tree-like structure. 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 [9].
- A small percentage of microservices are hot-spots in call graphs. 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.

- *Shutian Luo, Chengzhi Lu, and Kejiang Ye are with the Shenzhen Institute of Advanced Technology, CAS, University of CAS, Beijing 101408, China, and also with the Guangdong-Hong Kong-Macao Joint Laboratory of Human-Machine Intelligence-Synergy Systems, Guangzhou, Guangdong 441201, China. E-mail: {st.luo, cz.lu, kj.ye}@siat.ac.cn.*
- *Huanle Xu and Chengzhong Xu are with the University of Macau, Taipa, Macau 999078, China, and also with the Guangdong-Hong Kong-Macao Joint Laboratory of Human-Machine Intelligence-Synergy Systems, Guangzhou, Guangdong 441201, China. E-mail: {huanlexu, czxu}@um.edu.mo.*
- *Guoyao Xu, Liping Zhang, and Jian He are with the Alibaba Group, Hangzhou, Zhejiang 310052, China. E-mail: {yao.xgy, liping.z, jian.h}@alibaba-inc.com.*

Manuscript received 22 Jan. 2022; revised 6 Apr. 2022; accepted 30 Apr. 2022. Date of publication 12 May 2022; date of current version 26 July 2022.
This work was supported in part by the Key-Area Research and Development Program of Guangdong Province under Grant 2020B010164003, in part by the National Natural Science Foundation of China under Grant 62072451, in part by the Start-up Research Grant of University of Macau under Grant SRG2021-00004-FST, and in part by the Alibaba Group through Alibaba Innovative Research Program and Youth Innovation Promotion Association CAS under Grant 2019349.
(Corresponding authors: Chengzhong Xu and Kejiang Ye.)
Recommended for acceptance by Y. Yang.
Digital Object Identifier no. 10.1109/TPDS.2022.3174631

• Microservices can form highly dynamic call dependencies in runtime. In an extreme case, the same online service can have up to 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.

• Critical path is large-scale due to the dynamic call graph. 20% of online services contain more than 100+ critical paths during runtime. Such a large scale makes resource management based on critical paths infeasible in production environments [10].

The structure of call graphs affects the performance of online services significantly. Dynamic call graphs in an online service could be categorized into different classes based on their topologies. And we find that the end-to-end latency of an online service tends to be stable in call graphs of the same topology but varies significantly among different topologies. As such, topology-aware resource management is critical for microservice architecture. In addition, an online service could contain up to 70000+ critical paths, and the increase of critical path length will lead to a linear degradation in service performance.

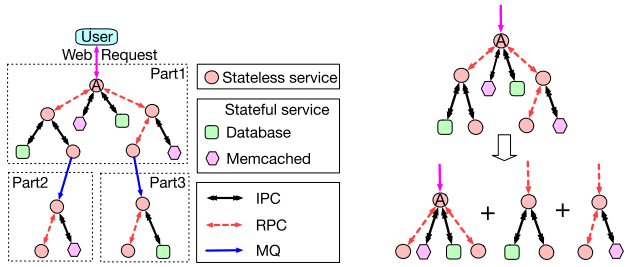
Microservice call rates (MCR) change periodically and have large fluctuations across the same times at different periods. Microservices serve a large number of requests from upstream microservices or users in runtime. Since users are usually diurnal, MCR fluctuates periodically throughout the day or week. Moreover, due to the dynamic and complex call dependencies between microservices, there are fluctuations across the same times at different periods. These non-negligible fluctuations will make the forecast of MCR in the future for each microservice more challenging.

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.

It is critical to manage resources based on heap memory in microservice architectures. Microservices run in containers, whose memory utilization is almost stable at runtime. Instead, we find that heap memory utilization of JAVA virtual machine (JVM) can well present the memory pressure. In addition, container memory utilization exceeds heap memory utilization by about 20% on average. This gap is caused by the garbage collection (GC) under which the released memory is not returned to the container. With this observation, it is more meaningful to focus on the JVM heap utilization directly for memory management.

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 the structural properties of microservice call graphs and the characteristic of call graphs for different applications (Section 3).
- We make a thorough characterization of microservice runtime performance, which provides deep insights into microservice scheduling (Section 4).
- We study the impact of interference and co-location on runtime performance, and provide optimization methods to enhance microservice performance (Section 5).
- We compare the difference between container memory and JVM heap memory in detail, which can be used to improve resource efficiency (Section 6).



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

Fig. 1. Illustrations of microservices.

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 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 and one downstream microservice. For ease of illustration, we sketch a call graph as shown in Fig. 1a. 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 downstream microservices in turn further call their downstream microservices.

Microservices can be categorized into two types, namely, stateless (a circle in Fig. 1a) and stateful (a rectangle or hexagon). Stateless services are isolated from state data while stateful services such as databases [11] and Memcached [12] 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 (IPC) [13], remote invocation [14], and indirect communication [8]. IPC usually happens between stateless and stateful microservices. Remote invocation such as Remote Procedure Call (RPC) [13] is a two-way communication under which a downstream microservice must return a result to its corresponding upstream microservice. By contrast,

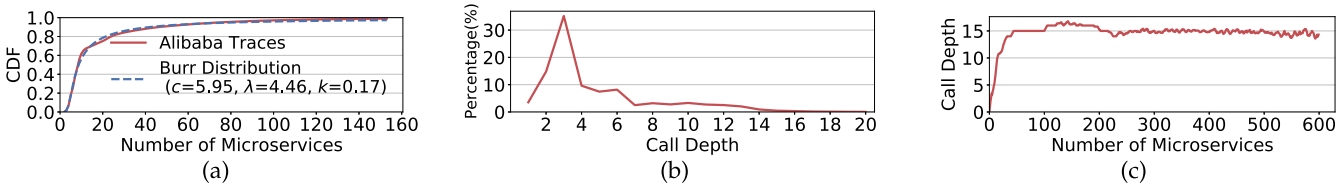


Fig. 2. Statistics of all microservice call graphs in Alibaba traces. (a) Cumulative distribution of the number of microservices in each call graph. For ease of presentation, we cut off the long tail to only count those numbers that are within the 99th percentile. (b) The distribution of call depth in all call graphs. (c) The maximum call depth (95th percentile) under a fixed number of microservices.

indirect communication such as Message Queue (MQ) is one-way only [13]. Under such communications, the upstream microservice sends a message to the third entity, which will persist the message for reliability, and the downstream microservice 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. 1a, 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 upstream microservice and all the downstream microservices it calls (Fig. 1b). 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. 1a is 5 (or it has 5 tiers).

The response time (RT) of a call is the length of the interval from upstream microservice calling its downstream microservice 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. 1a). 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 [15] to manage the bare-metal cloud [16] and, relies on the hardware-software hybrid virtio I/O system to enhance cluster performance and achieve better isolation between different services. 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.

2.2.2 Microservice Metrics

Alibaba makes use of the Application Real-Time Monitoring Service system to collect microservice traces [17], which is similar to Dapper [18]. A microservice usually runs in hundreds of containers. The microservice monitoring system collects system metrics for each container produced every minute and takes the average to record. These metrics range from hardware-layer, such as cache misses per kilo instructions, 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). The microservice monitoring system records in detail the call dependency between related microservices within a call graph. In detail, each record includes the information that a upstream microservice calls a downstream microservice via a specific interface in a specific call graph. From a single microservice point of view, the monitoring system also records all the calls (received from upstream microservices or sent to downstream microservices) related to each individual microservice. 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.

3 Overview 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 (Section 3.1). We then show how to apply graph learning algorithms to cluster the call graphs of each online service into multiple classes (Section 3.3). Last, we dissect microservice call graphs to focus on smaller components in each tier for generating new microservice benchmarks at a large scale (Section 3.4).

3.1 Characteristics of Overall 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. 2a, 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 1500+ microservices. This result indicates that the scale of existing benchmarks is far smaller than that

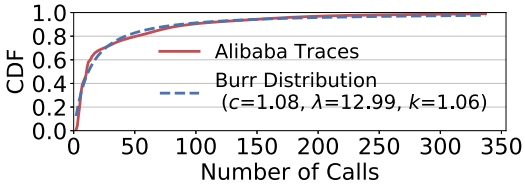


Fig. 3. Cumulative distribution of the number of calls in a call graph. For ease of presentation, we cut off the long tail to only count those numbers that are within the 99th percentile.

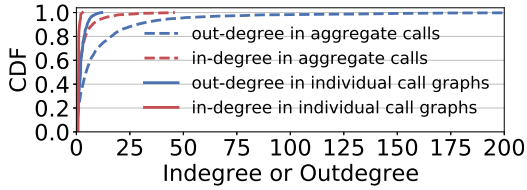


Fig. 4. Distribution of the degree of stateless microservices in individual graphs and aggregate calls.

in real traces [7], [8], [20]. Similarly, the number of calls in a graph also follows a Burr distribution, as shown in Fig. 3. Especially, more than 20% of call graphs include more than 50 calls, and the largest call graph can even consist of 54000 + calls. 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. 2b, 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 Memcacheds 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 graph can be as deep as 32 tiers and have an average depth of 4.84. As such, the call depth of a 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 [9]. 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 [21], [22]. 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 has a tree-like structure, and many of them can be structured only by a chain. As shown in Fig. 2c, 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. In this sense, microservice call graphs have far more scatter components than gather components. Here, we define scatter and gather components following the definition in [9]. In particular, if a microservice calls multiple downstream microservices in a call graph, such a set of microservices belongs to scatter components. By contrast, if a microservice is called by multiple upstream microservices in a call graph, such a set of microservices form gather components. The scattering property of microservice call graphs is different from that observed from traditional DAG graphs, which usually contain more 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 upstream microservice 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.

A small percentage of stateless microservices are hot-spots in call graphs. 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. Such microservices provide functions that many online services need. For example, payment microservices are shared by transaction-related services such as online shopping or train ticket ordering. 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.

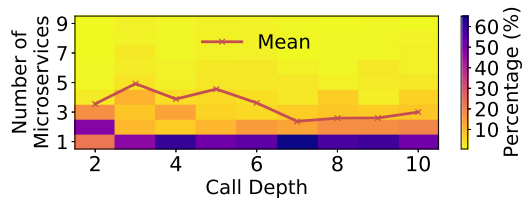


Fig. 5. The distributions of the number of microservices in different tiers.

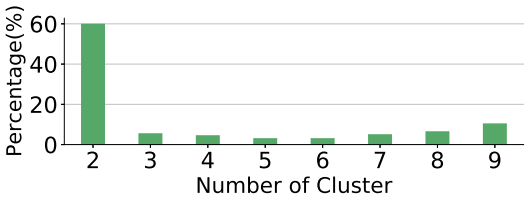


Fig. 6. The distribution of the number of classes of graph topologies in all online services.

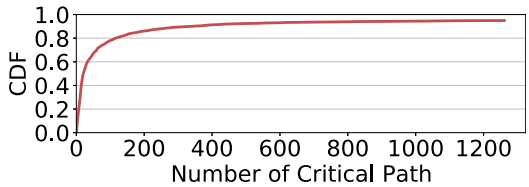


Fig. 7. The distribution of the number of critical paths under different online services.

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. 1a) and there 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 (Section 3.3). 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., [21], [22]).

Algorithm 1. Application Categorization

Input : S_i: service i,
M_i: the set of all microservices of S_i
T: the threshold for connected nodes
Output : A: application categories
1 Initiate an empty services graph G;
2 **for** *each services pair*(S_i, S_j) **do**
3 $percentage = Sharing(M_i, M_j)$;
4 **if** $percentage > T$ **then**
5 $G.addEdge(S_i, S_j)$;
6 $A = SpectralClustering(G)$;

An online service can consist of up to 70000 critical paths. Critical path is the call path in a graph that takes the longest processing time among all paths to handle users' requests [23]. Critical paths heavily impact the end-to-end performance and their characteristics could be leveraged to detect performance bottlenecks so as to, enhance resource management efficiency under microservice architecture. Previous works perform resource auto-scaling via considering all individual critical paths, e.g., [10], [24]. As such, their scalability is quite limited and can be hardly used in the production environment. As shown in Fig. 7, 20% of online services have 100+ critical paths in Alibaba and in an extreme case, some online services can contain up to 70000 critical paths. This large scale is due to that call graphs of the same service are highly dynamic and moreover, the same call graph can form a quite different call ordering among their contained microservices. In the meanwhile, we observe that 5% of critical paths could appear in more than 50% of call graphs. This observation implies that resource management solutions based on critical paths should focus more on those popular critical paths to achieve high efficiency.

3.2 Call Graphs Under Different Applications

After illustrating an overall picture of general call graphs, in this subsection, we carefully investigate how the characteristics of call graphs can vary across different applications. Our purpose is to provide insights on how to optimize the design of applications for performance improvement. In Alibaba, each application, such as an online shopping application and communication application, provides multiple different online services for users. Due to the lack of application identifiers in Alibaba traces, we first apply graph algorithms to recognize these applications based on the extracted connected components. Relying on these recognition results, we then study the characteristics of call graphs under different applications and explore how to develop applications using microservice architecture.

3.2.1 Application Categorization

Under conventional microservice architectures, microservices owned by each application can be shared among different services provided by this application. However, different applications usually do not share common microservices [25]. Nevertheless, there are many exceptions and it is difficult to achieve complete isolation between applications in the production environment. For instance, the payment microservice is shared among multiple applications in Alibaba. Due to this, it is impossible to separate applications based on their contained microservices only. By contrast, each individual online service only belongs to one application. As such, we choose to categorize application based on the service graph where each service is treated as one node, as described in Algorithm 1. We add an edge between two nodes in Step 9 if the fraction of their sharing microservices exceeds a certain threshold, such as 20%. Based on this built graph, we conduct *Spectral Clustering* [26] to divide it into multiple dense subgraphs where each subgraph represents an application. In Step 10, we range the number of candidate applications from 2 to 20, and choose the optimal number that can yield the highest Calinski-Harabaz score [27]. Although Calinski-Harabaz score prefers a smaller number of clusters [27], our categorization result matches well with the actual number of mainstream applications in Alibaba clusters.

Services in the same application are usually named following similar rules, such as starting with a specific prefix. Although the name of an application may be specified by multiple naming rules, we find that 50% of services in each

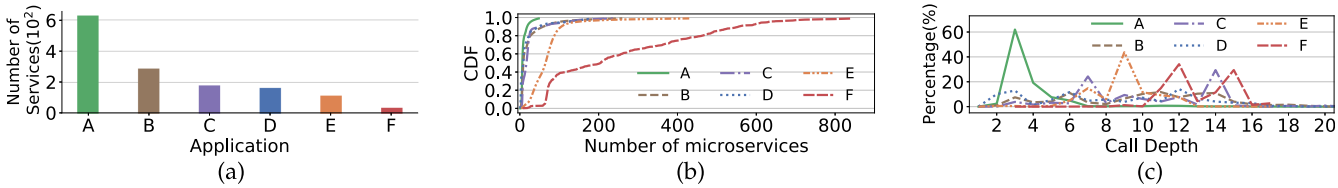


Fig. 8. Application-level statistics in Alibaba traces. (a) The distribution of the number of services under different applications. (b) The distribution of microservices in all call graphs for categorized applications. (c) The distribution of call depth under different applications.

application category share the same prefix. This further indicates our application categorization result is accurate. Moreover, we match the applications classified by Algorithm 1 to online applications in Alibaba according to their prefix and, investigate additional studies on application development in the sequel.

3.2.2 Analysis of Application Evolution

Applications are evolving gradually to provide higher throughput and serve more and more users. In this part, we characterize the statistical difference between applications that exist in Alibaba for a long time (long-term developed) and those that are recently deployed (short-term developed) to gain a better understanding of optimizing application performance.

Long-term developed applications provide 12× more online services compared to short-term ones. We count the number of services for each application categorized in Section 3.2.1 and depict the top six applications in Fig. 8a. Evolution periods of these six applications range from 15+ to 2 years in Alibaba. Long-term developed applications (e.g., A) provide 12 times more online services compared to short-term ones (e.g., F). Though both application A and F provide online shopping services, they serve different categories of goods and moreover, A is the core business in Alibaba and its architecture has been optimized for more than a decade. As such, application A tends to provide more and more services during its evolution and its contained microservice becomes lightweight gradually.

Long-term developed applications have tree-like structures of no more than four tiers and are more loosely coupled than short-term ones. We quantify the complexity of call graphs under different applications in terms of the number of microservices and call depth. Fig. 8b depicts the CDF distribution of the number of microservices in a call graph of different applications. It shows that most call graphs under application A have a small size and the average number is 9.5. As a comparison, call graphs of application F consist of 250 microservices on average. Furthermore, Fig. 8c shows that 60% of call graphs under application A have three tiers and less than 5% of them have more than four tiers. However, more than 50% of call graphs under application F have no less than 10 tiers. This indicates that, with the evolution of applications, tree-like structures of call graphs become simpler gradually. As a consequence, long-term developed applications can result in a much higher throughput to handle a larger number of service requests. In Alibaba, the number of requests processed by application A and B per unit of time is similar and is 600× (22×) larger than that of application F (E). In the meanwhile, we also study the out-degree and in-degree of each microservice in all graphs under different applications. As shown in Fig. 9, all these applications have quite similar out-degree distributions. However, microservices under long-term developed applications tend to have fewer upstream microservices. As quantified from Fig. 10, the average in-degree of microservices under application A is 12.5% less than that under application F. This implies that call graphs of long-term developed application have fewer gather components. In conclusion, long-term developed applications are more loosely coupled compared to short-term developed applications.

3.3 Graph Learning Algorithms

Call graphs of online services are dynamic in runtime, requests of the same online service shall involve different call graphs and their end-to-end latency varies a lot. As such, it is meaningful to quantify how the topology of a call graph can impact latency. However, studying each individual call graph is not scalable and we turn to study the latency performance within each class of similar topologies.

In this part, we present graph learning algorithms that are used to classify the call graphs of each microservice into different classes. The clustering results are used to analyze end-to-end latency of call graphs in the following sections. 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* [28]. *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 input to the deep neural network. By maximizing the mutual information conveyed in a training set, *InfoGraph* manages to

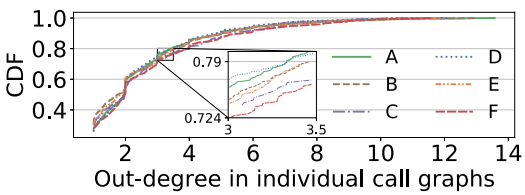


Fig. 9. Distribution of the out-degree of stateless microservices under different applications' call graphs.

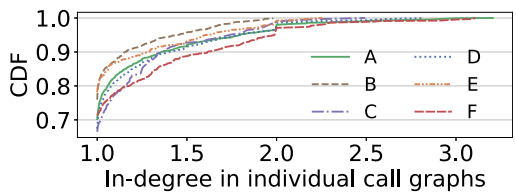


Fig. 10. Distribution of the in-degree of stateless microservices under different applications' call graphs.

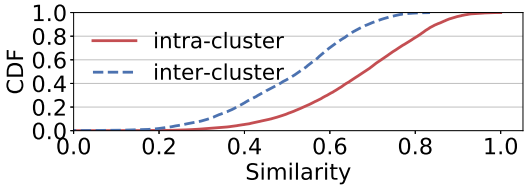


Fig. 11. Intra-cluster similarity versus inter-cluster similarity.

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. To find out the best clustering result, we first fix the number of clusters for Kmeans in the range [2,10] and compute the Silhouette score under this number. We then find the optimal result that yields the best score. As such, this clustering process is unsupervised and does not need true labels.

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 [29]. 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 2. In Step 2, 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. 11 ($m = 50$) 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 2. 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})

3.4 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 downstream microservice. 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

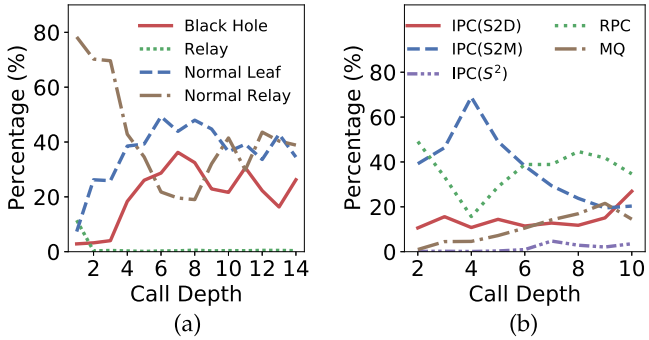


Fig. 12. Tier-level microservice statistics. (a) The percentage of different types of stateless microservices. (b) The distributions of communication paradigms in different tiers.

(or *normals*) shall call their downstream microservices with a certain probability. Moreover, the distribution of these three types varies over different tiers. As shown in Fig. 12a, 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 *normal* proceeds to call downstream microservices (i.e., the percentage of *normal relays* in Fig. 12a) 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.

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. 12b, 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. 12b. 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 Runtime Performance Characterization

Understanding the runtime performance of microservices is critical to ensure the quality of services. Since microservices run in hundreds to thousands of containers in Alibaba clusters and serve time-varying requests with highly dynamic call dependencies, several factors can affect the RT performance of microservices. In this section, we first analyze the characteristics of MCR in runtime. We then study the impact of MCR and graph topology on microservice response time performance.

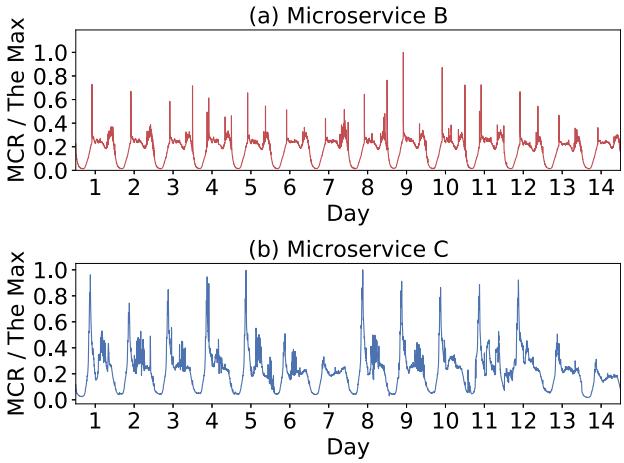


Fig. 13. The fluctuation of MCR for Microservice B and C within two weeks.

4.1 Microservice Call Rate

4.1.1 MCR Fluctuation Pattern

We analyze traces related to calls via RPC and quantify the number of calls sent to a microservice from its upstream microservices over time. Observe from Fig. 13 that, MCR of a microservice changes periodically over days, i.e., the MCR values are close to each other at the same time on different days. Take Microservice B for an example (i,e, Fig. 14). It provides an E-commerce service and users usually like to shop online when they are relaxed before lunch and going to sleep. As a result, the peak (circled by red circles) of B's MCR occurs before the noontime or late evening (i.e., 11:00 and 23:00) on each day. In addition, the behavior of B's MCR is the same across different days in a week since users tend to have the same online shopping habit no matter they are on weekdays or at weekends. By contrast, Microservice C provides services of enterprise communication and collaboration for employers to join intensively before they start work. As such, the peak MCR of C appears nearly at 9:00 am and this value shows a non-negligible difference between weekdays and weekends. By making use of this periodicity, it is possible to make an accurate prediction of the workload in data centers and thus make dynamic resource provisioning to maximize resource efficiency.

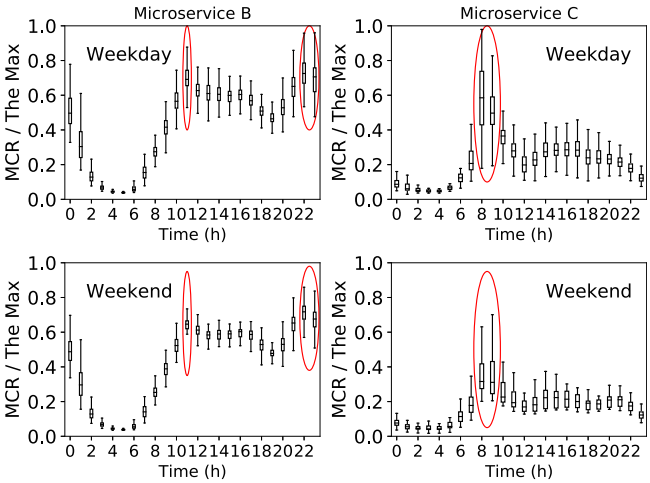


Fig. 14. The MCR of online microservices B and C changes over time within one day.

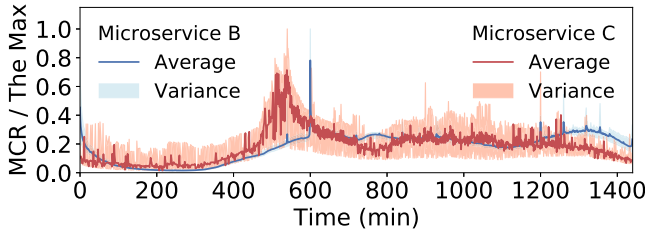


Fig. 15. The fluctuation of MCR within five days. The solid line denotes the average MCR generated in the same minutes of these five days, while the shallow area represents the variance.

Although MCR changes periodically over days, it can have large fluctuations across the same times on different days. Such fluctuations are due to the dynamic and complex call dependencies between microservices. We depict in Fig. 15 the fluctuations of B and C for an illustrative example. The max MCR of B (appears at 10:00 am) in a particular day is 40% higher than the average value of the maximum daily MCR within the following five days, and similar results can be observed on C. These non-negligible fluctuations will make the forecast of MCRs in the future for each microservice more challenging.

4.1.2 Simulation on Practical MCR

Existing works simulate and analyze online services from a queueing perspective with the assumption that the arrivals of service requests (MCR) follow a Poisson Process [7], [30], [31], [32], [33]. However, such a simple assumption is insufficient to handle complicated service arrival patterns in a production environment. To validate this argument, we adopt both the common Gaussian distribution and mixture models [34] such as the Gaussian Mixture Model (GMM) to fit the actual distribution of MCR [35]. For the fitting of Gaussian and Poisson distributions, we apply the method of Maximum Likelihood Estimation [36] to derive their parameters. By contrast, we use Bayesian inference [37] to obtain the posterior probability of GMM following the Dirichlet process [38]. We then get the parameters of GMM by maximizing the posterior probability.

We obtain the actual distribution of MCR from the traces and compare it with the other three fitted distributions, i.e., Gaussian and Poisson distribution as well as GMM. We depict the comparison result for two representative microservices in Fig. 16. It indicates that simple unimodal distributions do not fit well with the actual MCR distribution since the latter usually contains multiple peaks. In addition, we also quantify the KL divergence between the actual distribution and other fitting models for all microservices. As shown in Fig. 17a, Poisson distribution yields the worst fitting result and GMM model matches quite well with the ground truth for each microservice. In particular, the KL divergence under the GMM distribution is almost one for all microservices while more than 60% of microservices have a divergence value larger than two if the Poisson distribution is fitted. To get rid of this Poisson-distributed assumption on the call arrival process, it is more suitable to analyze the microservice response time based on a G/G/1 queue.

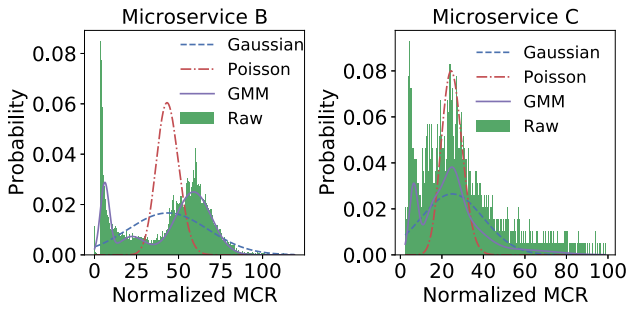


Fig. 16. The comparison between the fitted distributions and the actual distribution of MCR for microservices B and C. Since the Possion distribution only takes discrete random variables, we normalize the real MCR to take integers between 0 and 100.

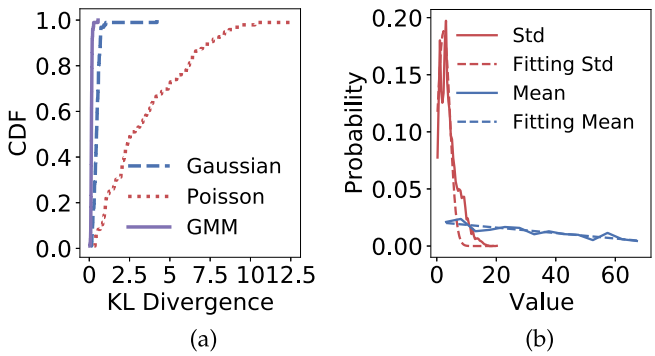


Fig. 17. (a) The cumulative distribution of the KL divergence between the fitted distribution and the actual distribution for all microservices. (b)The distribution of GMM parameters for all microservices, i.e., Mean and Standard Deviation (Std). The distribution of mean is fitted by a linear function $y = -0.00023x + 0.02$. And the distribution of Std is fitted by Gaussian distribution with $\mu = 2.3$ and $\sigma = 2.1$.

We also characterize the statistics of the mean and standard deviation for each fitted GMM distribution. As depicted in Fig. 17b, the standard derivation of GMM models used to fit each microservice MCR follows a Gaussian distribution. By contrast, the distribution of mean can be approximated using a linear function. With such information, one can generate microservice workloads that perform similar to production traces, which can be used to optimize system design for microservice architecture.

4.2 Performance Under Different MCR

MCR measures the number of calls received by a microservice in each minute per container. In this part, we characterize the relationship between MCR and several OS-level and application-level metrics to quantify the resource pressure in Alibaba clusters. And then, we study how MCR affect microservices performance in runtime.

Microservice call rates highly correlate with CPU utilization and Young GC but not with memory utilization. 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 sequences for each microservice. In Fig. 18, we plot the distribution of Spearman Correlations between MCR and multiple different metrics in both OS-level and application-level. As illustrated in Fig. 18, all microservices show a positive correlation between the

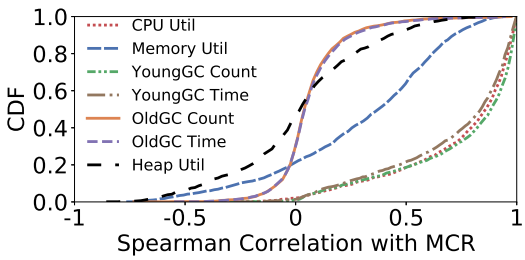


Fig. 18. The correlation between microservice call rate and different performance metrics.

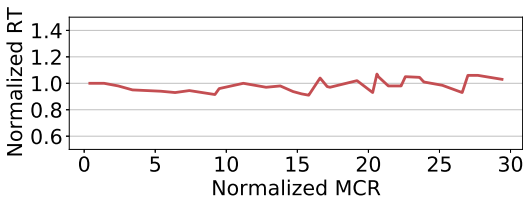


Fig. 19. RT performance under different normalized microservice call rates.

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 compared to memory utilization. A key reason behind this 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 [39], i.e., most containers exhibit steady memory but their CPU utilization varies in production clusters.

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 performs load balance among all containers, the resulted RT should only depend on the MCR per container, 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. 19. 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.

4.3 Performance Under Dynamic Call Graph

Call graphs are highly dynamic in runtime, which will affect end-to-end latency of online services significantly. In this

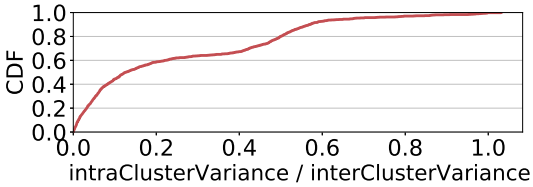


Fig. 20. Cumulative distribution of RT intra-cluster variance to RT inter-cluster variance ratios.

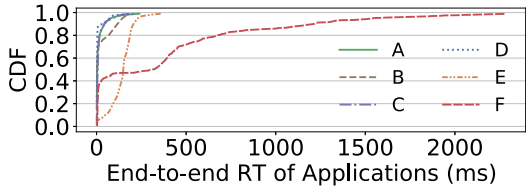


Fig. 22. End-to-end RT of requests under different applications.

part, we will study how graph structure can affect service performance and to what extent this impact is.

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 2 to cluster all the call graphs of each service into multiple classes. As stated in Section 3.3, 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. Fig. 20 depicts the cumulative distribution of the ratio of intra-cluster-variance to inter-cluster-variance for all services. It shows that this ratio is always less than 1, in other words, intra-cluster-variance is less than inter-cluster-variance. This implies that service requests with similar call graphs have similar end-to-end latency. In addition, more than 90% of online services have a small ratio (less than 0.6). This 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.

End-to-end latency of online services will increases linearly in the length of critical path. Since the response time of each request is dominated by the longest path processing time, we proceed to quantify how the characteristics of critical path can affect RT. Existing works also study the impact of critical path on microservice response time [10], [40], however, they do not quantify how such impact can vary across different critical paths and different services. As shown in Fig. 21, the end-to-end latency of online services will be almost stable when their critical paths have a length less than five. By contrast, when the length of critical paths exceeds 20, RT increases almost linearly with the length. When increasing the length from 20 to 80, the average RT performance of online services will be degraded by nearly two times. This can be used as a rough estimation of service RT given the topology of the call graph, which can be further used to configure SLAs when designing resource scheduling algorithms with performance guarantees. Furthermore, this observation is in line with the conclusion made in Section 3.2.2 that long-term developed applications have simple graph structures to achieve better performance. As illustrated in Fig. 22, the 95th percentile RT under long-term developed applications (such as A, B, C, and D) is less than 80ms, while this value becomes more than 1.5 seconds for short-term developed applications including E and F.

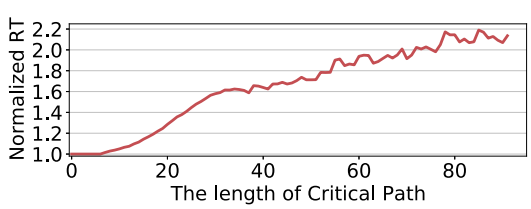


Fig. 21. RT performance degrades with the increase of the length of critical path. Normalized RT is the average ratio of the RT under each length of critical path to RT under the shortest critical path among all online services.

5 Performance Optimization

In this section, we show several factors including resource interference and communication overhead can greatly degrade service RT performance. Based on this observation, we further make several suggestions on optimizing microservice performance.

Balancing CPU utilization across different hosts can improve RT performance significantly. 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 which are deployed in different hosts. However, these requests are evenly distributed to all the instances of each downstream microservice. In this sense, the impact of downstream hosts can be treated as a constant when the utilization of this host changes. Although RT turns out to be stable with the increase of MCR in most microservices as revealed in Section 4.2, there still exists a small part of microservices whose RT can be impacted by MCR. Therefore, we collect all RTs of microservices with fixed MCR 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. 23 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 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

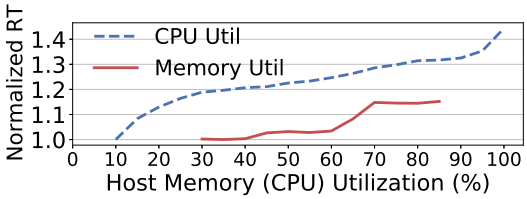


Fig. 23. Performance degradation due to resource interferences on the same physical host. X-axis represents the host CPU (memory) utilization while Y-axis quantifies the 75th percentile normalized RT among all microservices.

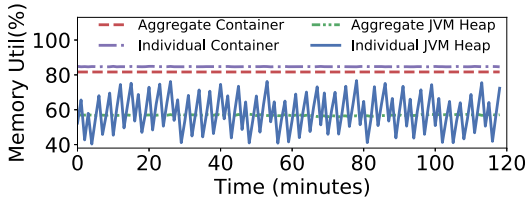


Fig. 25. The memory utilization of a representative microservice (including two aspects, i.e., both JVM heap memory and container memory) with a single container or a group of containers during a period of two hours.

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.

Colocation of dependent microservices could improve RT performance by 22% on average. Invocation between a pair of microservices is usually via RPC call or Message Queue. It can lead to a large communication overhead when many instances of these dependent microservices are located far away from each other. To investigate this, we quantify the difference in response time between colocated instances and non-colocated instances within the same pair of microservices. As illustrated in Fig. 24, co-location can reduce the response time by 22% on average, and 10% of colocations can even lead to an improvement up to 80%. It implies that, reducing communication overhead can greatly reduce microservice response time especially when the number of instances is large. This again validates the argument that, coupling the interfaces of dependent microservices together can help to optimize RT performance.

6 Resource Management

In this section, we investigate resource management policies under microservice architecture. In production clusters, each microservice is deployed with hundreds to thousands of containers, which are usually configured with different sizes of CPU and memory and managed by by various scheduling policies [?]. CPU utilization of containers reflects how long a user process occupies the slicing time, which is managed by the operating system. Due to this, CPU utilization of a running container is an instant metric. In addition, the reported CPU utilization from the operating system is quite sensitive to the type of workload that is being processed. As such, CPU utilization can well reflect the activity of microservices in real time, i.e., MCR, as described in Section 4.2. Consequently, we can directly tune CPU configuration for microservice containers based on the predicted MCR.

In contrast, memory utilization is usually high in production clusters [39], which has even become a new resource bottleneck at Alibaba [41]. Different from CPU resource, memory resource is kept occupied by a process before it is released by itself. As such, memory utilization is a long-term metric, which accumulates gradually in runtime. This explains why memory utilization shows a weak correlation with MCR (Section 4.2). In this sense, it is difficult to manage memory resources based on MCR directly.

Existing works mainly focus on CPU resource management [21], [42]. However, memory resource is more expensive than CPU resource [17] and is more challenging to deal with. As such, only tuning CPU configuration can easily lead to a waste of memory resources. In this paper, we investigate how to efficiently manage memory resources for microservices.

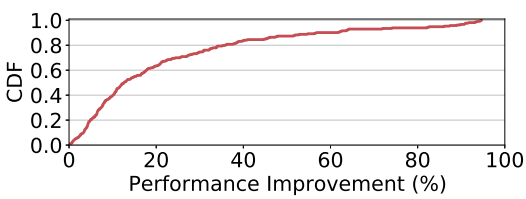


Fig. 24. Performance improvement via co-location of dependent microservices.

6.1 Resource Tuning Based on Heap Memory

More than 95% of microservice containers use JVM to manage their memory at Alibaba. Heap memory contributes a large part to the overall memory used by JVM during runtime. As shown in Fig. 25, for each container, the curve of its heap memory utilization has a jagged shape and the utilization ranges from 40% to 80%. When the heap memory usage exceeds the default threshold, i.e., 80%, it will trigger a garbage collection in JVM to free up the heap memory. However, the released memory pages are still marked by JVM as busy and will not be returned to the container due to the design mechanism of JVM. As such, even though the heap memory utilization of a container has a significant fluctuation, the memory utilization of the corresponding container keeps quite stable.

We can also observe from Fig. 25 that the average heap memory utilization of all containers present stable, and its value is very close to the median of a single container's heap memory utilization. In addition, there is a significant gap (more than 20%) between heap memory utilization and container memory utilization. These observations imply that, the cluster memory should be managed based on the heap memory at runtime instead of the container memory, since the latter can easily cause misleading.

6.2 Group-Level Memory Management

We adopt the average heap memory utilization of a group of containers to eliminate the fluctuations following the Law of Large Numbers [43]. We also range the group size for each microservice from 1 to 2000 to further study to what extent, the fluctuation of the memory utilization within a group is negligible. As shown in Fig. 26, the heap utilization fluctuates between 30% and 90% when the group only has one container. By contrast, when the group size increases to 400, the average

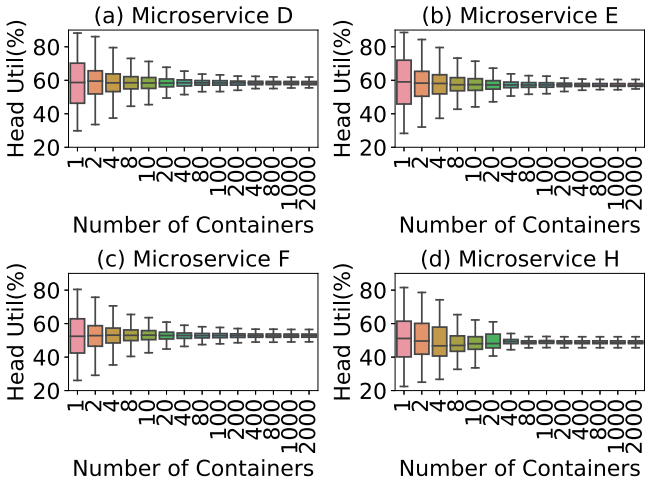


Fig. 26. The distributions of JVM heap utilization over time with respect to different number of containers.

utilization shall converge and the fluctuation is quite small. As such, one can take multiple samples of the running containers from a microservice at the same time, and compute the average heap memory utilization to infer the actual resource usage. By doing so, the memory size of each container can be scaled down substantially to save cluster resources. For example, the heap utilization of the microservice illustrated in Fig. 26a converges to 60% and thus the cluster scheduler only needs to allocate (a little bit more than) 60% of the original memory configuration to release the remaining resources for other services. To further strengthen this argument, we take samples of running containers from the whole cluster uniformly at random. More specifically, we investigate all microservices to evaluate their average memory utilization under different group sizes. We observe these microservices have similar performance in terms of heap memory utilization, as shown in Fig. 26. As the group size goes up, the average heap utilization does not change over time. In this case, the scheduler can tune memory resources for each microservice based on its average heap memory utilization within a large group.

7 Related Work

Microservice Benchmarks. Prior works use benchmarks, such as DeathStarBench [8], μSuite[7] and Acme Air [6] 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 [8]. Ueda *et al.* focused on the network overhead and explored how to reduce it under the microservice architecture [6]. 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 [14]. 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 [44]. 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 [45], [46], Alibaba Trace [9], [39], Azure Trace [47], [48], 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 [49]. 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 [50], 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., [39], [45], [47], [51]. However, they lack investigations on graph structures. Recently, Tian *et al.* conducted a comprehensive analysis of Alibaba DAG job traces [9]. 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. The computation process of analyzing production traces in a large scale involves heavy data read and write, which can be accelerated by compression-based techniques [52].

Performance Characterization of Online Services. Existing works also adopt Markov Chain to simulate call dependencies between different online services [30], [47], [53]. 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 [49], software such as socket or hardware such as NIC [22], 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 [10], [21], [22]. 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, the number of critical paths is large due to dynamic call graphs. 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. Moreover, our categorization on microservices application also provides useful implications on the software development and performance optimization of online applications.

Our study on microservice run-time performance provides several good implications on microservice scheduling and cluster resource management. In particular, the observation that RTs of online services highly depend on the call graph topology and performance bottleneck will change among microservices in different call graph topologies necessitate topology awareness for more efficient microservice schedulers, which however is quite challenging.

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.

Acknowledgments

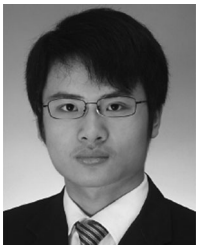
We would like to thank the anonymous reviewers for their thoughtful comments to this paper. Huanle Xu is co-first author. Shutian Luo and Huanle Xu contributed equally to this paper.

References

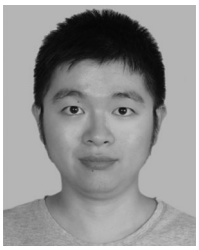
[1] S. Luo *et al.*, "Characterizing microservice dependency and performance: Alibaba trace analysis," in *Proc. ACM Symp. Cloud Comput.*, 2021, pp. 412–426.
[2] "CNCF," 2021. [Online]. Available: https://www.cncf.io/
[3] "Amazon web services," 2021. [Online]. Available: https://aws.amazon.com/
[4] "Google cloud," 2021. [Online]. Available: https://cloud.google.com/
[5] "Alibaba cloud," 2021. [Online]. Available: https://www.alibabacloud.com/
[6] T. Ueda, T. Nakaike, and M. Ohara, "Workload characterization for microservices," in *Proc. IEEE Int. Symp. Workload Characterization*, 2016, pp. 1–10.
[7] A. Sriraman and T. F. Wenisch, "μ suite: A benchmark suite for microservices," in *Proc. IEEE Int. Symp. Workload Characterization*, 2018, pp. 1–12.
[8] Y. Gan *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proc. Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2019, pp. 3–18.
[9] H. Tian, Y. Zheng, and W. Wang, "Characterizing and synthesizing task dependencies of data-parallel jobs in alibaba cloud," in *Proc. ACM Symp. Cloud Comput.*, 2019, pp. 139–151.
[10] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An intelligent fine-grained resource management framework for slo-oriented microservices," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2020, pp. 805–825.
[11] "Mysql," 2021. [Online]. Available: https://www.mysql.com/
[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. ACM Symp. Cloud Comput.*, 2010, pp. 143–154.
[13] A. S. Tanenbaum and M. Van Steen, *Distributed Systems: Principles and Paradigms*. Englewood Cliffs, NJ, USA: Prentice-Hall, 2007.
[14] A. Sriraman and T. F. Wenisch, "μtune: Auto-tuned threading for OLDI microservices," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 177–194.
[15] "Kubernetes," 2021. [Online]. Available: https://kubernetes.io/
[16] X. Zhang, X. Zheng, Z. Wang, H. Yang, Y. Shen, and X. Long, "High-density multi-tenant bare-metal cloud," in *Proc. Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2020, pp. 483–495.
[17] "Alibaba cloud product," 2021, [Online]. Available: https://www.alibabacloud.com/product
[18] B. H. Sigelman *et al.*, "Dapper, a large-scale distributed systems tracing infrastructure," Tech. Rep., 2010.
[19] A. Hakim, I. Fithriani, and M. Novita, "Properties of Burr distribution and its application to heavy-tailed survival time data," *J. Phys. Conf. Ser.*, vol. 1725, 2021, Art. no. 012016.
[20] X. Zhou *et al.*, "Poster: Benchmarking microservice systems for software engineering research," in *Proc. IEEE/ACM 40th Int. Con. Softw. Eng.: Companion*, 2018, pp. 323–324.
[21] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, "Sinan: ML-based and QoS-aware resource management for cloud microservices," in *Proc. Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2021, pp. 167–181.
[22] Y. Gan *et al.*, "Seer: Leveraging Big Data to navigate the complexity of performance debugging in cloud microservices," in *Proc. Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2019, pp. 19–33.
[23] C.-Q. Yang and B. P. Miller, "Critical path analysis for the execution of parallel and distributed programs," in *Proc. 8th Int. Conf. Distrib.*, 1988, pp. 366–367.
[24] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, and J. Mars, "GrandSLAm: Guaranteeing SLAs for jobs in microservices execution frameworks," in *Proc. 14th Eur. Conf. Comput. Syst.*, 2019, pp. 1–16.
[25] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Softw.*, vol. 35, no. 3, pp. 24–35, May/Jun. 2018.
[26] U. Von Luxburg, "A tutorial on spectral clustering," *Statist. Comput.*, vol. 17, no. 4, pp. 395–416, 2007.
[27] T. Caliński and J. Harabasz, "A dendrite method for cluster analysis," *Commun. Statist.-Theory Methods*, vol. 3, pp. 1–27, 1974.
[28] F.-Y. Sun, J. Hoffman, V. Verma, and J. Tang, "Infograph: Unsupervised and semi-supervised graph-level representation learning via mutual information maximization," in *Proc. Int. Conf. Learn. Representations*, 2020, pp. 1–16.
[29] S. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt, "Graph kernels," *J. Mach. Learn. Res.*, vol. 11, pp. 1201–1242, 2010.
[30] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi, "An analytical model for multi-tier internet services and its applications," in *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 33, no. 1, pp. 291–302, 2005.
[31] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal, "Dynamic provisioning of multi-tier internet applications," in *Proc. 2nd Int. Conf. Autonomic Comput.*, 2005, pp. 217–228.
[32] T. Wood, P. J. Shenoy, A. Venkataramani, and M. S. Yousif *et al.*, "Black-box and gray-box strategies for virtual machine migration," in *Proc. 4th USENIX Conf. Netw. Syst. Des. Implementation*, 2007, pp. 17–17.
[33] J. Cao, M. Andersson, C. Nyberg, and M. Kihl, "Web server performance modeling using an M/G/1/K* PS queue," in *Proc. 10th Int. Conf. Telecommun.*, 2003, pp. 1501–1506.
[34] B. Everitt, *Finite Mixture Distributions*, Berlin, Heidelberg, Germany: Springer, 2014.
[35] C. E. Rasmussen, "The infinite gaussian mixture model," in *Proc. Adv. Neural Inf. Process. Syst.*, 2000, pp. 554–560.
[36] H. White, "Maximum likelihood estimation of misspecified models," *Econometrica: J. Econometric Soc.*, vol. 50, no. 1, pp. 1–25, 1982.
[37] G. E. Box and G. C. Tiao, *Bayesian Inference in Statistical Analysis*, vol. 40, Hoboken, NJ, USA: Wiley, 2011.
[38] Y. W. Teh, "Dirichlet process," *Encyclopedia Mach. Learn.*, vol. 1063, pp. 280–287, 2010.
[39] Q. Liu and Z. Yu, "The elasticity and plasticity in semi-containerized co-locating cloud workload: A view from alibaba trace," in *Proc. ACM Symp. Cloud Comput.*, 2018, pp. 347–360.
[40] L. Zhao *et al.*, "Rhythm: Component-distinguishable workload deployment in datacenters," in *Proc. 15th Eur. Conf. Comput. Syst.*, 2020, pp. 1–17.

[41] J. Guo *et al.*, "Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces," in *Proc. IEEE/ACM 27th Int. Symp. Qual. Serv.*, 2019, pp. 1–10.
[42] H. Nguyen *et al.*, "AGILE: Elastic distributed resource scaling for infrastructure-as-a-service," in *Proc. 10th Int. Conf. Autonomic Comput.*, 2013, pp. 69–82.
[43] L. E. Baum and M. Katz, "Convergence rates in the law of large numbers," *Trans. Amer. Math. Soc.*, vol. 120, no. 1, pp. 108–123, 1965.
[44] T. Yu *et al.*, "Characterizing serverless platforms with serverlessbench," in *Proc. ACM Symp. Cloud Comput.*, 2020, pp. 30–44.
[45] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proc. ACM Symp. Cloud Comput.*, 2012, pp. 1–13.
[46] M. Tirmazi *et al.*, "Borg: The next generation," in *Proc. Eur. Conf. Comput. Syst.*, 2020, pp. 1–14.
[47] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms," in *Proc. ACM 26th Symp. Operating Syst. Princ.*, 2017, pp. 153–167.
[48] M. Shahrad *et al.*, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 205–218.
[49] H. Zhou *et al.*, "Overload control for scaling wechat microservices," in *Proc. ACM Symp. Cloud Comput.*, 2018, pp. 149–161.
[50] K. Veeraraghavan *et al.*, "Kraken: Leveraging live traffic tests to identify and resolve resource utilization bottlenecks in large scale web services," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 635–651.
[51] G. Amvrosiadis, J. W. Park, G. R. Ganger, G. A. Gibson, E. Baseman, and N. DeBardeleben, "On the diversity of cluster workloads and its impact on research results," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 533–546.
[52] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and X. Du, "POCLib: A high-performance framework for enabling near orthogonal processing on compression," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 2, pp. 459–475, Feb. 2022.
[53] A. Kamra, V. Misra, and E. M. Nahum, "Yaksha: A self-tuning controller for managing the performance of 3-tiered web sites," in *Proc. 12th IEEE Int. Workshop Qual. Serv.*, 2004, pp. 47–56.

Shutian Luo received the BSc degree in electrical engineering from Shenzhen University, in 2016. He is currently working toward the PhD degree with the Computer Application Technology, University of Chinese Academy of Sciences. His research interests include resource management in cloud computing. Shutian won the Best Paper award of SoCC2021.

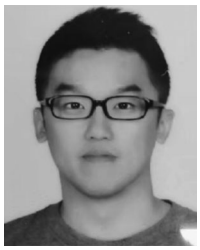


Huanle Xu (Member, IEEE) received the BSc (Eng) degree from the Department of Information Engineering, Shanghai Jiao Tong University (SJTU), in 2012, and the PhD degree from the Department of Information Engineering, The Chinese University of Hong Kong (CUHK), in 2016. He is currently an Assistant Professor with the Department of Computer and Information Science, University of Macau. His primary research interests include job scheduling and resource allocation in cloud computing, decentralized social networks, parallel graph algorithms, and machine learning. He is particularly interested in designing and implementing wonderful algorithms for large-scale systems and their applications using optimization tools and theories. Huanle is also a reception of 2021 ACM SoCC best paper award.



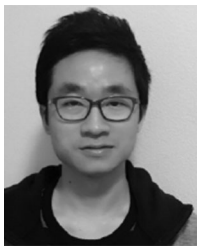
Chengzhi Lu received the BS degree in computer science from Wuhan University, Wuhan, in 2016, and the MEng degree in software engineering from Zhejiang University, Hangzhou, in 2018. He is currently working toward the PhD degree with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen. His research interests focus on co-location in cloud platform.

Kejiang Ye (Member, IEEE) received the BS and PhD degrees from Zhejiang University, in 2008 and 2013, respectively, and the joint PhD degee from The University of Sydney, in 2012 to 2013. He is currently a professor and the deputy director with the Research Center for Cloud Computing, Shenzhen Institute of Advanced Technology (SIAT), Chinese Academy of Sciences (CAS). Before joining SIAT, he was a post doctoral research associate with Carnegie Mellon University (CMU), from 2014 to 2015, and was a research fellow with Wayne State University (WSU), from 2015 to 2016. His research interests focus on the performance, energy, and reliability of Cloud computing, Big Data and Network systems . He is a member of SPEC Research Group and is also a member of ACM, CCF and CIE.



Guoyao Xu received the BS degree from Xidian University, Xi'an, China, in 2011, and the master's and PhD degrees from Wayne State University, in 2013 and 2019, respectively. He is currently a expert engineer with Alibaba Cloud. His research interests include datacenter operation optimization.

Liping Zhang received the bachelor's and PhD's degrees in eletrical engineering from Tsinghua Univeristy, in 1999 and 2004, respectively. He is currently a principal engineer with the Alibaba group. His research interests include system research, in particular cluster management.



Jian He received the bachelor's degree from University of Electronic Science and Technology of China, and the master's degree from Brown University. He is a staff engineer with Alibaba and his interest is in distributed systems related areas.

Chengzhong Xu (Fellow, IEEE) received the PhD degree from The University of Hong Kong, in 1993. He is currently the chair professor of computer science and the dean with the Faculty of Science and Technology, University of Macau. Prior to this, he was with the faculty with Wayne State University, USA, and the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, China. He has published more than 400 papers and more than 100 patents. His research interests include cloud computing and data-driven intelligent applications. He was the Best Paper awardee or the Nominee of ICPP2005, HPCA2013, HPDC2013, Cluster2015, GPC2018, UIC2018, AIMS2019, and HPBD&Is2020. He also won the Best Paper award of SoCC2021. He was the Chair of the *IEEE Technical Committee on Distributed Processing* from 2015 to 2019.