Generating Microservice Graphs with Production Characteristics for Efficient Resource Scaling

Fanrong Du*
Shanghai Jiao Tong University
Shanghai, China
dufanrong@sjtu.edu.cn

Jiuchen Shi*
Shanghai Jiao Tong University
Shanghai, China
The Hong Kong Polytechnic University
Hong Kong, Hong Kong
shijiuchen@sjtu.edu.cn

Quan Chen†
Shanghai Jiao Tong University
Shanghai, China
chen-quan@cs.sjtu.edu.cn

Pu Pang
Shanghai Jiao Tong University
Shanghai, China
pangpu@cs.sjtu.edu.cn

Li Li
Shanghai Jiao Tong University
Shanghai, China
lilijp@cs.sjtu.edu.cn

Minyi Guo
Shanghai Jiao Tong University
Shanghai, China
guo-my@cs.sjtu.edu.cn

Abstract

A production microservice application can have multiple services with varying call graphs, and a microservice may be shared across different call graphs. Improving resource efficiency in such complex applications requires proper benchmarks, but production traces are often too large to be used in experiments. To this end, we propose a Service Dependency Graph Generator (**DGG**) that comprises a *Data Handler* and a *Graph Generator*, to generate service dependency graphs of benchmarks that incorporate production-level characteristics from traces. The data handler constructs fine-grained call graphs with dynamic interface and repeated calling features from the trace, and then clusters these call graphs based on the topological and invocation types. The graph generator uses a random graph model to simulate real microservice invocations, generating multiple call graphs and merging them into small-scale service dependency graphs with production-level characteristics. Case studies show that DGG-generated graphs closely resemble real traces in topology. Moreover, evaluating the same resource scaling strategy, DGG-generated benchmarks show a 24.2% increase in resource efficiency while real trace benchmarks show a 27.3% increase, proving that DGG can yield similar results in evaluating resource management strategies to real-world traces.

CCS Concepts

• **Computer systems organization** → **Cloud computing**.

Keywords

Micorserivces, Characteristics, Graph generator

ACM Reference Format:
Fanrong Du, Jiuchen Shi, Quan Chen, Pu Pang, Li Li, and Minyi Guo. 2025. Generating Microservice Graphs with Production Characteristics for Efficient Resource Scaling. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3721145.3725761

*Fanrong Du and Jiuchen Shi contributed equally to this work.
†Quan Chen is the corresponding author.

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3725761

1 Introduction

Microservice architecture is an emerging paradigm in modern software development, which decomposes the monolithic service into loosely coupled, independently deployable microservices [3, 36]. A service composed of microservices can be denoted as a Directed Acyclic Graph (DAG) where the vertices and edges represent the microservices and call dependencies, respectively [22, 23]. To guarantee the Quality of Service (QoS) [11, 39, 49], the computing resources of each microservice need to be scaled with the load change.

As observed from open-source microservice traces [15, 22, 23], a service often exhibits intricate *call graphs* with diverse

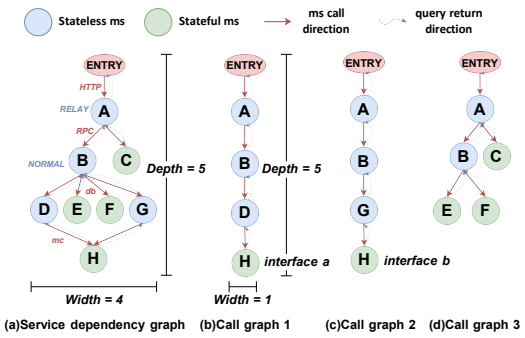


Figure 1: An example service dependency graph and its three call graphs.

vertices and edge types. Figure 1 shows the service dependency graph of a service from the Alibaba trace and its three call graphs. All call graphs of a service form its *service dependency graph*. Queries accessing the same service may go through part of the microservices based on user-specified requirements, forming different call graphs [22, 23]. Moreover, there are stateless and stateful microservices (e.g., business logic and databases) [47], and diverse communication modes (e.g., Remote Procedure Call (RPC) and HTTP) [22, 23]. Both stateful and stateless microservices have a set of Application Programming Interfaces (APIs) for upstream microservices, and some microservices are shared by call graphs [15, 24]. For instance, microservice H has two interfaces *a* and *b* that are called by different call graphs in Figure 1.

While the traces are often too large to be reproduced in research, it is important to build benchmarks[1] that reflects the characteristics of production microservice applications [12, 13, 38, 42, 45, 50]. Current benchmarks are often abstracted from small-scale applications, lacking commonly existing production characteristics like dynamic interfaces [12, 13, 38, 42]. For example, Google Online Boutique [13] has 5 call graphs and 11 microservices, with an average depth of 2.8 and only 3 microservices featuring dynamic interfaces. Nevertheless, observed from the real Alibaba trace, there are more than 40 call graphs per service, with the maximum call depth exceeding 15. Moreover, 48.8% of microservices have more than two interfaces, and a service can include more than 500 microservices. The assistant tools that allow researchers to customize benchmarks [10, 33] also fail to capture realistic production characteristics.

Several automatic tools are also proposed to generate call graphs from production characteristics [22, 23]. They are not sufficient for two major defects. As for the first defect, the characteristics of the production microservices that guide the generation of benchmarks are incompletely analyzed. Although previous studies have examined the call graph characteristics of microservices, they still lack a thorough analysis of service dependency graph variations, microservice inter-relationships in call graphs, and various interfaces for individual microservices. These characteristics are crucial for investigating microservice resource scaling, as they can lead to heterogeneous computing resource demand of microservices. In this case, current graph generation tools fail to capture the full complexity of production microservices.

As for the second defect, the services generated by existing tools have large deviations from reality. The service dependency graph is generated based on the overall trace statistics, without perceiving the variations between different services. Moreover, the generated call graphs have little inter-relationship, which cannot reflect microservice sharing features among different call graphs. Also, inside a specific call graph, the repeated calling and sibling relationship among microservices are not reflected. From the perspective of individual microservices, the characteristics of various interfaces are ignored in these tools.

To address the above issues, we propose a *Service Dependency Graph Generator (DGG)* that includes a *Data Handler* and a *Graph Generator* to generate the service dependency graphs of benchmarks that include the production characteristics[2]. It is proposed on the basis of three new observations from the thorough analysis of production traces. 1) Upstream microservices may have repeated calls to the downstream microservices, and microservices have dynamic interfaces called by different call graphs. 2) The dependency graphs of different services vary significantly in topology and invocation patterns. 3) Microservices typically invoke a set of child microservices, and the probability of a microservice calling a specific children set is influenced by its sibling microservices.

Based on above three observations, the design requirements of DGG is to capture repeated callings and dynamic interfaces, address the variability among different service types, as well as represent the invoked microservices as the children set while consider the effect of sibling microservices. Therefore, DGG's data handler first constructs fine-grained call graphs from production traces with the dynamic interfaces and repeated callings, and merges them to form dependency graphs. Then, it clusters the dependency graphs into different categories based on their topological and invocation features. Based on the organized data, the graph generator creates random graph models to represent microservice calls as children set invocations that are influenced by the sibling microservices. It then generates call graphs based on these models and merges them to form the final dependency graph.

We also conduct detailed case studies on using DGG to generate service dependency graphs, as well as investigating

[1]A benchmark is a service composed of microservices.

[2]DGG is open-sourced via https://github.com/dufanrong/DGG.

the similarity in evaluating resource management efficiency between DGG-generated benchmarks and real-trace based benchmarks. In more detail, we use DGG to generate different types of service dependency graphs with associated call graphs, and investigate the similarities between the generated and real call graphs in terms of topology. The results show that the call graphs generated by DGG are similar to real-world ones. Moreover, we validate the similarity of evaluating the same resource scaling strategy efficiency on DGG-generated benchmarks and benchmarks for real traces. Results show that this strategy increases resource efficiency by 24.2% and 27.3% on average while ensuring the QoS on the DGG's benchmarks and real-trace benchmarks, respectively. These results show that DGG' benchmarks yields similar results in evaluating the efficiency of resource management strategies to benchmarks based on real-world traces.

The major contributions of this paper are as follows.

- **In-depth analysis of production microservice traces.** The analysis reveals novel observations of production microservices that motivate the design of DGG.
- **The design of the dependency graph generator that incorporates production characteristics.** DGG first constructs fine-grained graphs from production trace and clusters them into different categories. Then, DGG generates service dependency graphs for each service type based on novel random graph models.
- **Case studies on benchmark generations and microservice resource scaling efficiency evaluation.** We validate the topology similarity between DGG's generated graphs and the real-world ones, as well as the similarity between DGG-generated benchmarks and real-world benchmarks on evaluating efficiency of resource management strategies.

2 Related Work

In this section, we discuss related work on cloud trace analysis, microservice benchmarks, and call graph generators.

Cloud trace analysis: Some studies analyzed the runtime performance [4, 7, 8, 32] or resource usage [35] of lots of types of cloud workloads or production clusters. These works did not aim at the microservice architecture. Moreover, some other studies focused on the characteristics of microservice call graphs [15, 22, 23, 44, 48]. However, they lacked in-depth exploration of the variations in service dependency graphs, the relationships between microservices in call graphs, and the dynamic interfaces of individual microservices.

Microservice benchmarks: Numerous benchmarks were developed for research on microservice resource scaling [12, 13, 42, 45, 50], but they failed to reflect the realistic characteristics of real-world microservices [34]. Specifically, DeathStarBench [12] had 3 services, with 4, 3, and 1 call graphs, respectively. μsuit [42] consisted of four different services, each with only two microservices, whereas production services usually consist of dozens to hundreds of microservices [19, 22]. Moreover, these benchmarks lack rich communication modes of realistic microservices that significantly affect latency and resource management [18, 26]. For example, μsuit only adopted gRPC for inter-microservice communication. At last, these benchmarks include little sharing characteristic among microservices, which commonly exists in production environment [24, 34].

Some previous works also proposed assistant tools to support flexible customization of microservice scale, topology, and behaviors by developers [10, 33], but they cannot capture production microservice characteristics automatically.

Call graph generation tools: Luo et al. proposed a call graph generator based on the distribution of production microservices [22, 23]. However, this generator modeled the overall distribution of all services, and thus cannot capture the variations in different service dependency graphs. The generated call graphs are also not inter-connected, making it difficult to construct a complete service dependency graph. Moreover, this tool failed to consider several production microservice features, including microservice sharing, communication modes, and sibling effects among microservices.

3 Background and Terminology

Service Dependency Graphs and Call Graphs. The user queries may go through part of the microservices in the service dependency graph based on user characteristics, forming different call graphs. Microservices accessed by multiple call graphs are referred to as shared microservices in this paper. For instance, an e-commerce recommendation service can recommend products based on two filter conditions, including the price and rate microservices. The user queries that select price filtering, rate filtering, and both price and rate filtering will form three different call graphs, respectively. The three call graphs share the price and rate microservices.

Inside a service dependency graph or call graph, there is an entry microservice for receiving user queries, e.g., nginx [30]. For a specific call, the triggering and called microservices are the upstream and downstream microservices (UM and DM) [22, 23]. An UM typically calls a set of DMs, referred to as the *children set*, and the DMs with the same UM are considered as *sibling microservices*. In addition, a single UM can have multiple children sets. In Figure 1(d), A is the entry microservice that receives queries via HTTP. B and C are sibling microservices, as well as they are DMs of A (which is the UM) and also construct the children set of A.

The topological characteristics of the service dependency graph or call graph mainly encompass depth and width. Depth is the longest path from the entry microservice to any

other microservice, while width is the maximum number of microservices at any given layer of depth. In Figure 1(b), the depth of this call graph is 5 and the width is 1.

Microservices Types. There are stateful and stateless microservices in production [47]. Stateful ones are typically databases and caching middlewares like mongodb [29] and memcached [27]. Stateless ones are mostly related to business logic [22]. Based on their interactions with downstream microservices, stateless microservices are further categorized into RELAY, LEAF, and NORMAL [22]. RELAY must have downstream microservices, LEAF no longer calls others, and NORMAL will call others with a certain probability.

Communication Modes. Microservices mainly communicate through Inter-Process Communication (IPC) [46], Remote Procedure Call (RPC) [41], or Message Queue (MQ) [12]. IPC typically occurs between stateless and stateful microservices [22], like calling mongodb and memcached. RPC is a type of synchronous communication, in which the requester needs to wait for the reply of the responser with a blocking mode. MQ is an asynchronous method where microservices communicate via message queues like Kafka [20] and RabbitMQ [16]. In this model, there is no need for immediate response, enabling non-blocking communications.

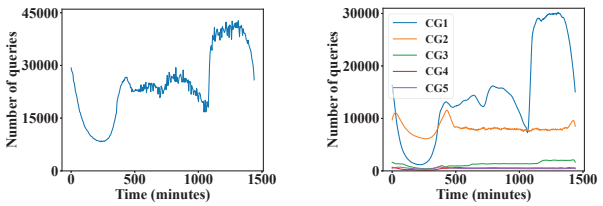
4 Microservice Trace Analysis

In this section, we analyze the characteristics of the service dependency graphs, call graphs, and call characteristics of microservices in the production clusters of Alibaba and Meta. These are the only two open-source microservice traces, representing typical cloud applications (Taobao and Facebook).

4.1 Overview of Alibaba and Meta Traces

The Alibaba trace dataset v2022 [2] has over 20 million call graphs involving more than 17,000 microservices across ten clusters over 13 days. In this trace, each service is identified by a service ID, while each user query is tracked with a unique trace ID. Each pair of microservice calls is identified by a rpcID, including details about the upstream microservice (UM), the downstream microservice (DM), the type of communication (rpctype), the interface of the downstream microservice invoked, etc. [22, 23]. We randomly select the data of 80% services in this trace for trace analysis in this section and for evaluating the similarities between DGG-generated and real call graphs in Section 6.1. A service refers to all data in the trace associated with the same service ID. Moreover, we use the remaining 20% of the data to validate our trace observations in resource management in Section 6.2.

The Meta trace dataset [28] features a microservice topology with 18,500 active services and over 12 million service instances. This trace includes information on the service type, call depth, maximum width, the set of downstream microservices (DM set) invoked by each microservice, the number of upstream microservice (UM) invocations to the children set in a single query, and other relevant metrics [15].



(a) Number of queries over time. (b) Queries of different call graphs.

Figure 2: Total number of queries and queries of different call graphs for the service *S_130831269* in the Alibaba traces.

4.2 Characterizing Dependency Graphs

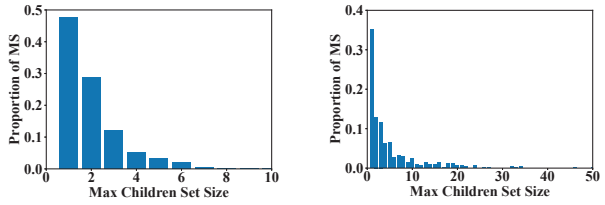
Since the Meta trace [28] lacks complete dependency graph information, we analyze the characteristics of the Alibaba trace [2] in this subsection. We have two major observations. 1) Production service dependency graphs are dynamic over time. These graphs encompass many different call graphs, with their proportions shifting dynamically. 2) There is significant variation in topological characteristics and microservice invocation patterns across service dependency graphs.

4.2.1 Dynamic Service Dependency Graphs. Service dependency graphs in production exhibit dynamic behaviors, with call graphs changing over time. The occurrences of services in Alibaba traces follow a long-tailed distribution, where a small number of service dependency graphs contribute most of the query counts. On average, there are over 45 call graphs in each service dependency graph.

For a service, the number of queries per minute changes over time, as do queries accessing different call graphs. For example, Figure 2 shows the total query number and the number of queries accessing each call graph over time, for the service with the highest number of queries (S_130831269) in Alibaba traces. We observe that obvious variations in both the total queries and queries accessing different call graphs.

4.2.2 Topological Differences in Dependency Graphs. Different service dependency graphs in production exhibit significant differences in topology and invocation patterns.

In terms of the topology, the depth and width of these graphs vary greatly. For the top 20 services with the most queries, the 4th and 7th top services have a depth of 2 while 8th and 11th top services reach a depth of 6. The width also varies, for example, the 2nd and 19th top services have a width less than 2, while the 4th and 7th top services exceed 14. In terms of invocation patterns, specific patterns



(a) Alibaba trace. (b) Meta trace.

Figure 3: Distribution of the children set sizes for microservices.

are unique to certain dependency graphs. For instance, the self-invocation pattern (i.e., a microservice calls itself) is only present in 1st, 8th, and 11th top services. Moreover, unlike the other services, the 1st, 4th, and 7th top services do not include calls to databases or memcached.

4.2.3 Insights from Characterizing Dependency Graphs. Service dependency graphs in production clusters are dynamic, each consisting of multiple call graphs. To utilize production microservice data for research, it is necessary to extract features to generate simulated service dependency graphs. Given the large topological and invocation pattern differences, we should categorize them using clustering methods to study features and construct benchmark for each category.

4.3 Characterizing Call Graphs

Both Alibaba and Meta traces are used for analysis, and the key observations are as follows. 1) Microservices overlap among the children sets of different call graphs. 2) Some microservices within a children set may be called multiple times. 3) The probability of a microservice invoking different children sets is influenced by its sibling microservices.

4.3.1 Characteristics of Children Sets. Figure 3 shows the distribution of children set sizes in both traces. The children set size is the number of different microservices contained within a single children set. In the Alibaba trace, the sizes of the children sets range from 1 to 10, while those in Meta can be as large as 50. There are two major characteristics of microservice calls to children sets.

Overlap in Children Sets: Many microservices are shared by queries from different call graphs, with significant overlap in the children sets, meaning the same microservice is invoked in multiple call graphs. In the Meta trace, 92.2% of microservices appear in different children sets. Similarly, in the Alibaba trace, this overlap rate is 77.1%.

Repeated Calls: Repeated calls refer to a UM calling the same DM multiple times within a specific query, typically driven by application logic [23]. For example, in Google Online Boutique [13], the "currency" microservice is called twice for displaying product prices: first to fetch available currencies, then to convert prices. In the Meta trace, 20.4% of repeated calls occur within children sets, and this value is 16.2% for the Alibaba trace. Notably, database and memcached microservices exhibit a much higher rate of repeated calls at 52.5% in the Alibaba trace. Tables 1 provide detailed statistics on the number of repeated calls in both datasets.

Table 1: Statistics for the number of repeated calls to each microservice in the traces

Meta Trace	Min	Median	Mean	P99	Max
Total	2	17	210	2,339	2,392

Alibaba Trace	Total	Database	Memcached	Others
Min	2	2	2	2
Median	3	3	5	3
Mean	16	13	30	12
P99	374	**198**	**469**	75
Max	1080	**397**	**537**	1080

4.3.2 Sibling Set Influence. The sibling microservices of an UM (i.e., the sibling set) can influence its probability of invoking its children sets. Given a service dependency graph $G(V, E)$ with microservices V and invocation relationships E. G can include multiple call graphs $CG_i(V_i, E_i)$. For each call graph CG_i, microservice u has a sibling set S_i (nodes that share the same UM as u) and a children set C_i (nodes that are invoked by u).

The $P(u \rightarrow C_i)$ represents the probability that microservice u invokes the children set C_i. It is calculated as the proportion of queries where u invokes C_i out of all queries where u invokes any possible children set. $P(u \rightarrow C_i \mid S_j)$ represents the probability that microservice u invokes the children set C_i given that the sibling set of u is S_j. When there exist $P(u \rightarrow C_i \mid S_j) \neq P(u \rightarrow C_i)$, we say that the probability of microservice u invoking the children set is influenced by the sibling set. In extreme cases, we might have $P(u \rightarrow C_i \mid S_j) = 1$, meaning the sibling set completely determines the children set that u invokes.

In the Alibaba trace, 55.7% of the services are influenced by sibling sets, where the probability of their microservices invoking DMs is related to sibling sets. In these services, 92.7%, 64.8%, and 76.2% of the microservices are influenced at call depths of 3, 4, and 5, respectively. This variation relates to the number of microservices at each depth: more microservices at the same depth lead to more sibling set combinations impacting children set calls. The sibling set influence is determined by the nature of the microservice architecture. A user query triggers inter-microservice calls between related

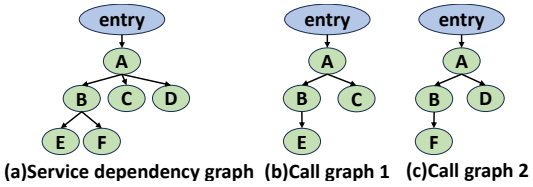


Figure 4: An example of the sibling set influence.

microservices, meaning those in the same call graph are functionally related. In some services, this functional correlation is more obvious, demonstrating sibling set influence.

Sibling effect is common in microservice clusters. We consulted authors of Meta trace paper [15] on Github, and they confirm that it is common in Meta cluster. This finding creates opportunities for more accurate call graph generation and microservice architecture optimization. First, modeling microservice calls as a probability model influenced by sibling sets better reflects the real-world production cluster scenarios, rather than considering only the two-level calls between UM and DM. Second, microservices with significant sibling set influence can be reorganized into larger microservices. In Figure 4, microservice B calls E when its sibling set is $\{C\}$ and calls F when its sibling set is $\{D\}$. This suggests that B, C, and E are tightly coupled and can be merged to reduce inter-microservice call overheads. Similarly, the microservices B, D, and F can also be merged.

4.3.3 Insights from Characterizing Call Graphs. To generate more realistic simulated call graphs from production traces, we should model downstream microservices as children sets and consider richer call information, including repeated calls and sibling set influence. Moreover, both fine-grained tracing and resource management of repeated microservice calls are essential for more effective microservice resource scaling.

4.4 Characterizing Individual Microservices

Since the Meta trace [28] lacks detailed microservice interface information, we analyze individual microservices using the Alibaba trace [2]. We have two key observations: 1) Significant variation exists in the number of interfaces provided by microservices with different communication patterns. 2) Microservice interfaces are called by different call graphs.

4.4.1 Types of Microservice Interfaces. We observe that the *memcached* and *http-called* microservices have more interfaces, while other microservices are simpler and have fewer interfaces. Most of the microservices in the Alibaba are relatively simple with 88.31% of them have less than 10 interfaces.

The number of microservice interfaces varies significantly across different communication modes. Table 2 presents statistics on the number of microservice interfaces under various communication modes. The number of interfaces for the

Table 2: Number of interfaces across different communication modes

	db	http	mc	mq	rpc
Min	1	1	1	1	1
Median	1	1	2	1	2
Mean	2	**11303**	31	3.0	9
P99	12	**14731**	26	39	74
Max	22	**1355296**	2728	76	89

HTTP-type microservices is notably higher. To know the reasons, we consulted the authors of the Alibaba trace regarding the HTTP interfaces. They informed us that the number of interfaces in an HTTP microservice (e.g., entering microservice) is often correlated with the parameters involved, which can lead to a high interface count, particularly when multiple parameters are present.

4.4.2 Interface Calling Patterns. We observe that microservices may be shared by call graphs from different user queries, and these different call graphs may call different interfaces of the microservice. In the Alibaba trace, there are five main patterns of call graphs calling the interfaces of microservices.

a) Some microservices are only called by the same call graph, which calls the same interface of the microservice each time. b) Microservices called by the same call graph may also call different interfaces. c) Different call graphs call the same interface of the microservice. d) Multiple call graphs share the microservice and each call graph calls a different interface of the microservice. e) For a microservice, some call graphs call the same interface, and the other call graphs call different interfaces.

4.4.3 Insights from Characterizing Individual Microservices. We should consider different interface calling patterns in the model of individual microservices when generating simulated service dependency graphs from production traces. Moreover, it is beneficial to conduct fine-grained tracing and resource management of different microservice interfaces, to improve resource scaling efficiency.

5 Dependency Graph Generation with DGG

In this section, we introduce the overview of DGG, followed by the design details and theoretical analysis of DGG's model.

5.1 Overview of DGG

As shown in Figure 5, we design and implement a **Service Dependency Graph Generator (DGG)** to generate microservice benchmarks that simulate realistic characteristics of services in production cluster traces. DGG consists of a *Data Handler* and a *Graph Generator*. The data handler is responsible for organizing the large amount of data in the

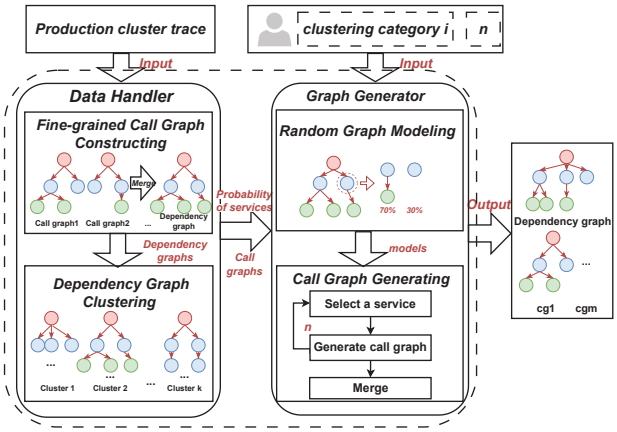


Figure 5: Design overview of DGG.

original production cluster trace. Based on the organized data from the data handler, the graph generator takes the clustering category (i) and call graph generation times (n) from the user, and then adopts two steps to generate the service dependency graphs with a series of call graphs.

Based on observations from trace analysis, microservice calls often exhibit repeated invocations, dynamic interfaces, and significant differences among services. Therefore, DGG's data handler adopts two steps to effectively address these characteristics. First, the *fine-grained call graph constructing* constructs precise call graphs from the production traces and merges them into different dependency graphs. Then, the *dependency graph clustering* classifies dependency graphs of services into different categories based on their topological and invocation characteristics.

To generate simulated call graphs that resemble realistic ones, DGG's graph generator first constructs a random probabilistic model for each service. Based on the observations in Section 4.3, it models downstream microservices as children sets and incorporates the impact of sibling sets, rather than modeling relationships solely as a two-level structure between UM and DM. Using these models, the *Call Graph Generating* process generates n call graphs and merges them into the service dependency graph.

DGG works in the following steps. 1) The data handler constructs precise call graphs from production traces and merges them into different dependency graphs of services. 2) The data handler then clusters the dependency graphs into different categories based on their topological and invocation characteristics. It also obtains the probability of occurrence of each service in each category. 3) With user inputs of the service category i and the number n of call graph generated, the graph generator builds a random graph model for each service in the category i. 4) Based on the random models, the graph generator then generates the call graphs n times, with

UM	UM_interface	DM	DM_interface
entry	none	A	func1
A	func1	B	func2
A	func1	B	func2
A	func1	B	func3

trace data

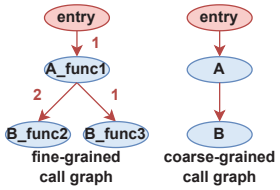


Figure 6: An comparison example of fine-grained and coarse-grained call graphs.

each time selecting a model of a service according to their probabilities of occurrence. 5) At last, the graph generator merges all call graphs to form the service dependency graph.

DGG is implemented in Python, and we use *GraKeL* library [40] to measure the similarity between dependency graphs. The statistical code lines of DGG are about 3000.

5.2 Data Handler

In this subsection, we introduce the design of fine-grained call graph constructing and dependency graph clustering.

5.2.1 Fine-grained Call Graph Constructing. Prior works [22, 24] construct call graphs as directed acyclic graphs, with vertices representing microservices and edges representing invocation relationships. However, this method fails to capture the repeated calls and dynamic interfaces common in production microservices. To obtain fine-grained call graphs, we represent microservice invocations triggered by queries using weighted directed graphs. We define vertices by combining the microservice name with the interface being called. The edges not only indicate invocation but also use weights to represent the number of times a DM is repeatedly called. Figure 6 shows fine-grained and coarse-grained call graphs generated from the same trace data. The fine-grained call graphs includes the representation of repeated calls and dynamic interfaces compared to coarse-grained ones. For each service in the trace, we construct all its fine-grained call graphs and merge them into the service dependency graph.

Since repeated calls and dynamic interfaces also impact microservice resource usage, this call graph constructor can be utilized for online microservice resource scaling to enhance resource allocation efficiency. We will use this scaling strategy in Section 6.2 to validate the similarity between DGG-generated benchmarks and real-world benchmarks on evaluating the resource efficiency.

5.2.2 Dependency Graph Clustering. To cluster the service dependency graphs by topological features and invocation properties, we use the Graph Kernel method [9, 40] to measure the similarity between dependency graphs, and then use K-means [1, 14] to cluster all the dependency graphs. The inputs to the Graph Kernel include the adjacency matrix of dependency graphs, the microservice labels (e.g., db,

memcached, normal), and the communication modes (e.g., http, rpc, mq), and the K value representing the clustering category number. The K value is selected with the highest silhouette coefficient, which is a metric to measure the clustering quality. Each cluster groups services with similar topological and call characteristics, enabling users to choose specific clusters for generating simulated graphs.

Adopting the above clustering method on the Alibaba trace [2], we categorize all the service dependency graphs into 6 categories. We use Graph Kernel method to quantify the similarity of different service dependency graphs both within each cluster and across different clusters. We find the intra-cluster similarity is 3.4X higher than the inter-cluster similarity, proving the accuracy of our clustering method.

5.3 Graph Generator

In this subsection, we introduce the process of building random graph models, and the algorithms for generating call graphs and dependency graphs using these models.

5.3.1 Random Graph Modeling. The variables used in the random graph modeling are shown in Table 3. For each service S in the trace dataset, we establish a random graph model $G_s(V_s, E_s)$ based on its real dependency graph $G(V, E)$.

The V represents the set of vertices in S's service dependency graph, where each $v \in V$ is a triplet representing the microservice's name, the interface being called, and its label, as *(ms_name, interface, label)* . The label set is L = *{database, memcached, normal, relay, leaf}*. The label of a microservice is determined by simple rules. If the communication mode is "db" and "mc", the labels are "memcached" and "database", respectively. For other communication modes, we examine whether the microservice continues to call other microservices. If it calls others, the label is "relay". If it never calls others, the label is "leaf". Otherwise, the label is "normal".

The E denotes the set of edges of the dependency graph of S, where each edge is represented as a tuple (u, v, w, t). u and v represents the source and destination microservices, w indicates the number of times v is repeatedly called, and $t \in T$ = *{http, rpc, mq, mc, db}* represents the communication modes, which can be directly obtained from the trace data.

We model the probability of each children set being called by a microservice. For a microservice u in S, let C_u represent all possible children sets that u may call. Each children set $C \in C_u$ consists of vertices v and the corresponding call edges e. Therefore, each children set includes the microservice name, interface, label, the number of repeated calls, and the communication mode. An empty children set indicates u does not call any other microservices. Let CG denote the set of all call graphs for S, where each call graph $cg_i \in CG$ has an associated occurrence count $count(cg_i)$.

Table 3: Variables used in Section 5.3

Variable	Description
S	A real service in the dataset
V	Collection of all microservices in S with microservice names, interfaces, and labels
E	Invocation edges in S with UM, DM, number of invocations, and communication mode
L	Set of all possible microservice labels
T	Set of all possible communication modes
CG	Set of all call graphs of S
cg_i	The i-th call graph of S
u	A microservice in S
C_u	Set of all children sets of u in CG
C	A children set of u with UM, number of calls, and communication mode
d	Depth of u's call in the call graph
s	Sibling set of u
count(cg_i)	Number of times cg_i is queried in S

Considering the sibling set influence, and the impact of microservices' call depth on children set calls, we characterize this influence using the conditional probabilities. We first define an indicator variable to indicate whether microservice u calls a children set under the conditions of a given depth and sibling set in a call graph as:

$$I(u \to C, s, d, cg_i) = \begin{cases} 1 & \text{if vertex } u \text{ calls children set } C \text{ at depth } d \\ & \text{with sibling set } s \text{ in call graph } cg_i, \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

Based on this, the probability that u calls children set C given u's sibling set s and depth d can be calculated in:

$$P(u \to C \mid s, d) = \frac{\sum_{cg_i \in CG} I(u \to C, s, d, cg_i) \cdot count(cg_i)}{\sum_{C' \in C_u} \sum_{cg_j \in CG} I(u \to C', s, d, cg_j) \cdot count(cg_j)} \tag{2}$$

The time to establish a random graph model for a real service depends on the dependency graph scale. In our experiments, the average model establishment time is 4ms.

5.3.2 Call Graph Generation Algorithm. Based on the established random graph model, the call graph generation process is as follows. 1) Initialization: The graph starts with the vertex *entry* as the beginning of the query. 2) Vertex and edge addition: For each vertex u labeled as *normal* or *relay*, determine its sibling set s and depth d. Based on Equation 2, choose a children set C and add it to the graph. For each (v, w, t) in C, add edge (u, v, w, t) to the graph. 3) Repeat for each new vertex u until no further vertices can be added.

Algorithm 1 shows the process of generating a call graph. The output is a call graph G stored as a list of call relationships. Each element in the list has the structure *(um_depth, UM, DM, weight, compara)*, where um_depth represents the

depth of UM in the call graph, UM and DM denote the respective vertices, *weight* indicates the number of times DM is repeatedly called, and *compara* is the communication mode. A queue Q stores vertices that are not handled. Initially, G is empty, and $(1, (entry, none, relay))$ is pushed into Q (line 3).

In the main loop, the algorithm pops a vertex and its depth (d, UM) from Q (line 5). If UM is labeled *relay* or *normal*, it determines the children set based on random graph model (lines 7-8). For other labels, the algorithm skips to next iteration. For each target vertex in the children set, it generates a new call relationship edge, adds it to G, and pushes the new vertex into Q (lines 9-12). The process continues until Q is empty, then returns the call graph G.

Algorithm 1 Call Graph Generator

Output: G: A call graph stored as a list of call relations

```
1:  Q ← queue to temporarily store vertices without generated DM
2:  G is initialized as an empty list
3:  Push (1, (entry, none, relay)) into Q
4:  while Q is not empty do
5:      (d, UM) ← Q.pop()
6:      if UM.label == relay or normal then
7:          s ← get_sibling_set(UM)
8:          targets ← children_set(UM, d, s)
9:          for target ∈ targets do
10:             (DM, weight, t) ← (target.v, target.w, target.t)
11:             G.add(d, UM, DM, weight, t)
12:             Q.push(d + 1, DM)
13:     else
14:         continue
15: return G
```

Algorithm 1 has a time complexity of $O(|V| \cdot k)$, where $|V|$ is the total number of vertices in the call graph, and k is average size of the children sets. It uses an list to store edges of the generated graph, with a space complexity of $O(|E|)$, where $|E|$ is the total number of edges in the call graph. In our experiments, the average time to generate a call graph is 0.7 ms. We use actual microservice names from the dataset, effectively managing scenarios with multiple parents, depths, or shared microservices across call graphs.

After all call graphs are generated with the user-specified n times, DGG merges them to form a dependency graph.

5.4 Theoretical analysis

In this subsection, we analyze the probability distribution similarity in topology between call graphs generated by DGG and those observed in the real-world trace dataset. We examine three aspects: the width (microservices number) at each layer, the depth of the call graph (total layers), and the total number of microservices in the call graph.

We give a sketch for the proofs of main results in this section, and the full proof can be found in https://github.com/dufanrong/Theoretical-Proof.

Since our probabilistic model (Eq.(2)) ensures the children set sizes in generated and actual call graphs match, we can first prove the width distribution is the same at each layer. Then, by using the fact that a call graph has exactly h layers only if all microservices at depth h have empty children sets, we can prove the depth distribution also matches. Finally, as the microservice number in a call graph is the sum of the microservices number (i.e., width) at each layer (i.e., depth), we can prove the total count of microservices is the same. The corresponding three propositions are as follows.

Proposition 5.1. *Let M_k represent the vertex number at the k-th level of the call graph. We denote $P_{gen\_width}(M_k = m_k)$ as the probability that the k-th level of the call graph has m_k vertices according to a random graph model, and $P_{real\_width}(M_k = m_k)$ as the probability that the k-th level of the real call graph has m_k vertices. Then,*

$$P_{gen\_width}(M_k = m_k) = P_{real\_width}(M_k = m_k) \tag{3}$$

Proposition 5.2. *Let H denote the total depth of a call graph. Let $P_{gen\_depth}(H = h)$ denote the probability that a generated call graph has depth h, and $P_{real\_depth}(H = h)$ denote the probability of depth h in the real dataset. Then,*

$$P_{gen\_depth}(H = h) = P_{real\_depth}(H = h) \tag{4}$$

Proposition 5.3. *Let N denote the total number of vertices in a call graph, $P_{gen\_num}(N = n)$ denote the probability that the generated call graph has n vertices, $P_{real\_num}(N = n)$ denotes the probability that the real call graph has n vertices. Then,*

$$P_{gen\_num}(N = n) = P_{real\_num}(N = n) \tag{5}$$

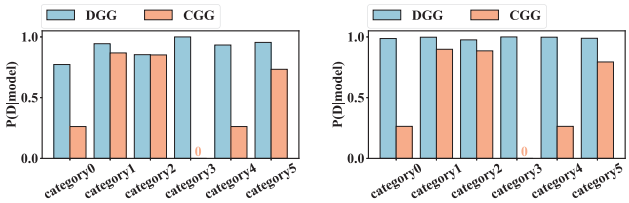
6 Case Studies

In this section, we first utilize DGG to generate service dependency graphs and evaluate their topology similarity with real-world ones. Then, we validate the similarity between DGG-generated benchmarks and real-world benchmarks from the traces in investigating resource management efficiency.

6.1 Graph Similarity of DGG

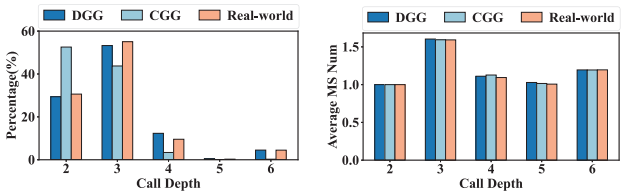
In this subsection, we first evaluate the graph topology similarity between DGG to real-world traces, followed by comparisons of statistical features like width, depth, and microservice number. Then, we conduct an ablation study on considering the sibling set effect and evaluate DGG's effectiveness on corner cases. Finally, we show DGG's generality on other benchmarks and traces.

6.1.1 Investigation Setup. Since the Meta trace [28] lacks detailed microservice call information, the Alibaba trace [2]



(a) Topology types of call graphs. (b) Frequencies of call graphs.

Figure 7: Topological similarity between the call graphs from traces and the call graphs generated by DGG and CGG, respectively.



(a) Percentages of Call graphs. (b) Average microservice number.

Figure 8: Call graph percentages and average microservice number under different depths of the DGG, CGG, and real-world dataset, respectively.

is adopted in this subsection. We compare DGG to a state-of-the-art microservice call graph generator (CGG) [22]. CGG computes layer width and assigns labels to microservices based on the probability distribution of the overall trace dataset to generate a call graph. According to the six clustering categories determined by DGG (Section 5.2.2), we construct six service dependency graphs, each with 100,000 call graph generations by using DGG and CCG, respectively.

In this subsection, We compare call graph similarity instead of directly comparing service dependency graphs, as the call graphs represent each query execution path which can capture dynamic interactions among microservices. We use $P(D \mid model)$ to quantify topological similarity between the generated and real-world call graphs. It denotes the proportion of generated call graphs structurally identical to real-world ones under different models (DGG or CGG). Two call graphs are identical if their node names, labels, edges, edge weights, and communication modes match exactly. We calculate $P(D \mid model)$ based on the distinct topology types and frequencies of the call graphs. For instance, when generating 5 call graphs, two of them have the same topology type A and the other three have the same topology type B. This results in two types of call graphs, and the frequencies of the first and second types are 2 and 3, respectively.

Moreover, we quantify the statistical feature similarity by using Jensen-Shannon (JS) divergence, which is proved to be effective in comparing two probability distributions [5]. Smaller JS divergence indicates closer distributions.

6.1.2 Graph Topology Similarity. With the generated call graphs of DGG and CGG, Figure 7 shows the $P(D \mid DGG)$ and $P(D \mid CGG)$ in each clustering category and the overall call graphs in terms of call graph types and frequencies.

We observe that, as for distinct types of generated call graphs, the similarity of call graphs generated by DGG is high across both each cluster category and the overall dataset, with an average value of 90.9% for the six types. In contrast, the call graphs generated by CGG exhibit lower similarity to the real graphs, with an average value of 49.6% for the six types. As for the frequencies of generated call graphs, the average accuracy of DGG is higher at 99.1%, while the accuracy of CGG is 51.7%. In terms of both the types and frequencies of generated call graphs, DGG shows higher similarity than CGG, generating more accurate call graphs.

Specifically, in cluster category $category3$, $P(D \mid CGG)$ is 0%. This is because the call graphs of services in this cluster category exhibit the repeated call feature to microservices, but CGG fails to capture this pattern. Instead, CGG generates call graphs where each microservice is called only once, missing the critical repeated call relationships present in the actual traces. Thus, every call graph generated by CGG does not match the real-world ones.

DGG properly clusters the call graphs into different types and considers multiple key characteristics like microservice sharing and repeated callings during the generation process, thus having a similar topology similarity. However, CGG generates all call graphs according to the overall dataset and does not comprehensively consider various characteristics.

6.1.3 Statistical Feature Similarity. Since the percentage of call graphs in depths over 6 is less than 10^{-5} in the Alibaba trace, we show call graphs with depths less than or equal to 6 in this subsection. Moreover, the percentage of call graphs with microservice numbers greater than 14 is less than 10^{-6}, so we also show call graphs with microservices less than 14.

Figure 8(a) shows the call graph percentages under different depths of the DGG, CGG, and the real-world dataset, respectively. DGG's call graphs closely match the distribution of real-world call graphs. In contrast, CGG shows a higher percentage at depth 2 and lower percentages for depths greater than 2, indicating early termination in the generated call graphs. The reason could be that CGG has a higher probability of generating microservices labeled "leaf", "memcached" and "db" that do not continue to invoke DMs. As statistics, the JS divergence of call graph percentage distributions between the DGG and real-world dataset is 0.034, whereas the value for CGG is 0.193.

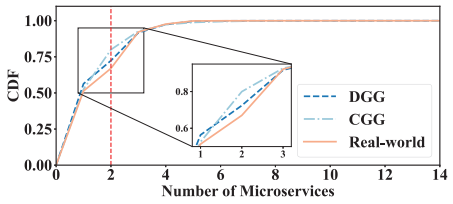


Figure 9: Cumulative call graph percentage distribution of the microsevice number. The red dotted line highlights that, for the same number of microservices, the DGG curve is closer to the Real-world curve than the CGG curve.

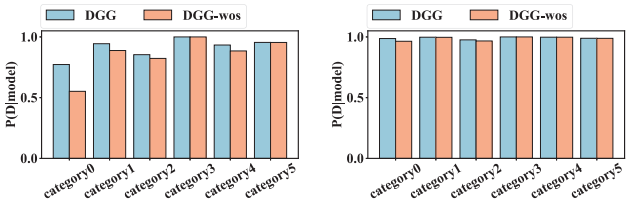
Figure 8(b) shows the average number of microservices (i.e., layer width) under different depths. DGG and CGG are both relatively close to real-world dataset. This is because CGG can also capture the distribution of the number of microservices at different depths in the real call graph during the modeling process. DGG's distribution is a little closer to real-world call graphs compared to CGG, with a JS divergence of 0.003 for DGG and 0.004 for CGG.

Figure 9 shows the cumulative call graph percentage distribution of microservice number of DGG, CGG, and real-world dataset, respectively. We can observe that DGG's curve closely matches the curve of real-world, while CGG's generated call graphs have fewer microservices. The JS divergence between DGG and real-world call graph percentage distributions is 0.053, while the value for CGG is higher as 0.146.

6.1.4 Ablation Study of the Sibling Set Influence. To evaluate the effectiveness of considering the sibling set as a conditional probability in generating call graphs (Equation 2), we conduct an ablation study to compare DGG and DGG-wos. DGG-wos is a variant of DGG in which the sibling set is excluded as a conditional probability in the modeling process. Instead, DGG-wos generates downstream microservices solely based on the upstream microservices and the call depth during the call graph generation process.

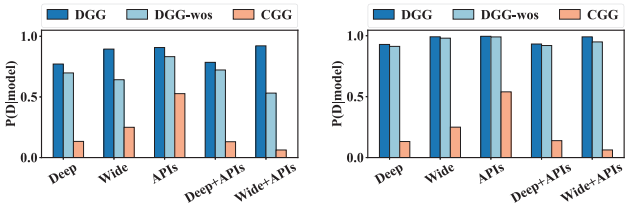
Figure 10 shows the $P(D \mid \text{model})$ of the call graphs generated by DGG and DGG-wos with the real-world call graphs from traces. For both the overall set of call graphs and each service cluster category, the call graphs generated by DGG are more similar to the real call graphs than those generated by DGG-wos. As for distinct types of generated call graphs, the average accuracy of the six types is 90.9% for DGG, while the value of DGG-wos is 85.1%. As for frequencies of generated call graphs, the average values of DGG and DGG-wos are 99.1% and 98.5%, respectively.

Specifically, for clustering category *type*3, call graphs generated by DGG and DGG-wos reach a 100% similarity with real ones. This is because the topology of the call graphs for



(a) Topology types of call graphs. (b) Frequencies of call graphs.

Figure 10: Topological similarity between call graphs from traces and call graphs generated by DGG and DGG-wos, respectively.



(a) Topology types of call graphs. (b) Frequencies of call graphs.

Figure 11: Topological similarity between the real trace corner call graphs and the call graphs generated by DGG, DGG-wos, and CGG, respectively.

this category is very simple, with a call depth of only 3 and a width of only 2, and all simply calling Memcached.

The accuracy gap in the call graphs generated for the *type*0 cluster by DGG and DGG-wos is the largest, with $P(D \mid \text{DGG}) = 77.23\%$ and $P(D \mid \text{DGG-wos}) = 55.2\%$ for distinct types, $P(D \mid \text{DGG}) = 98.7\%$ and $P(D \mid \text{DGG-wos}) = 96.4\%$ for frequencies. This is because the call graphs for this category of services have more complex topologies. With an average width of 3.4, compared to the overall average width of 1.6, these wider call graphs exhibit more intricate sibling microservice relationships.

DGG utilizes a sibling set as a condition for probability calculation, achieving higher accuracy in generating call graphs. In contrast, DGG-wos, which omits sibling relationships, results in lower accuracy. The difference between DGG and DGG-wos is more pronounced in corner call graph cases, which we will discuss later. According to state-of-the-art work from production [6], the tail latency gap between simple and complex call graphs averages 20× across varying loads, highlighting the importance of studying microservice resource scaling in corner cases.

6.1.5 Similarity for the Corner Call Graphs. The corner call graphs can significantly impact the end-to-end performance of services, such as deep call graphs leading to increased query latency. These call graphs are essential for studying

resource scaling in microservices. Therefore, we evaluate DGG's effectiveness in generating corner call graphs. To obtain corner call graphs for the trace dataset, we filter the trace data to extract the following types for evaluations: ① *Deep*: the call graphs with a depth greater than 6. ② *Wide*: the call graphs with a width greater than 3. ③ *APIs*: the call graphs invoke more than 2 interfaces of a microservice within the same service. ④ *Deep+APIs*. ⑤ *Wide+APIs*.

Figure 11 shows the similarity between the call graphs generated by DGG, DGG-wos, and CGG with the real call graphs in the above cases. The average accuracy of the five types of corner cases for call graph types is 85.6%, 68.5%, and 22.1% for DGG, DGG-wos, and CGG, while the average accuracy for the call graph frequencies is 96.8%, 95.0%, and 22.5% for DGG, DGG-wos, and CGG, respectively. DGG still achieves high accuracy in generating call graphs, even in corner cases, demonstrating its generality. Moreover, CGG remains the least accurate, and DGG-wos also performs worse than DGG without considering the sibling set effect.

For in-depth analysis, in the *APIs* scenario, the call graphs often involve microservices calling multiple interfaces of the same microservice within the same children set. Since both DGG and DGG-wos use the children set, rather than individual microservices with its interface, both can effectively handle such situations. Moreover, in the *Wide* and *Wide+APIs* scenarios, the call graphs are relatively wide and shallow, and DGG achieves higher accuracy in generating these graphs. This is because DGG uses sibling relationships as conditional probabilities, which allows it to more effectively handle cases where microservices have multiple siblings. By contrast, DGG-wos, which neglects the sibling effect, shows decreased accuracy. At last, in the *Deep* and *Deep+APIs* scenarios, as the call depth increases, the number of iterations in the probabilistic model also increases, resulting in a slight decrease in the accuracy of the generated call graphs.

6.1.6 DGG's Generality on Other Traces and Benchmarks. In this subsection, we show DGG's generality when adapting to other traces and benchmarks. To the best of our knowledge, there are currently only two available microservice traces (Alibaba and Meta traces [15, 22]), and Meta trace lacks fine-grained call information. Therefore, we adopt an alternative approach using the hotel reservation (HR) and social network (SN) applications in DeathStarBench [12] to generate traces. HR and SN support 4 and 3 types of queries, which refer to 4 and 3 types of distinct call graphs, respectively. Corresponding to default configurations in the source codes of DeathStarBench, we generate the traces at a Query-Per-Second of 500 for 200 seconds, with the proportions of 0.6:0.3:0.05:0.05 for the 4 types of queries of the HR and 0.6:0.3:0.1 for the 3 types of queries of the SN. We name these two traces as *H-Trace* and *S-Trace*, respectively.

Table 4: Comparisons of generated call graph quantity

	HR				SN		
	CG1	**CG2**	**CG3**	**CG4**	**CG1**	**CG2**	**CG3**
Real	60127	30006	5017	4850	59627	30263	10110
DGG	60084	29937	5058	4921	59430	30163	10077
CGG	1160	831	767	4572	19497	19803	417

Then, we use DGG and CGG to generate call graphs based on the *H-Trace* and *S-Trace*, respectively. Table 4 shows the generated call graph quantity of DGG, CGG, and the real-world values in the traces. DGG generates 4 call graphs for HR and 3 call graphs for SN, all of which are topologically identical to the real call graphs. Moreover, the generated quantity of DGG for all call graphs is also very close to the quantity of the real-world ones. This shows that DGG is a flexible tool that can generate dependency graphs and call graphs with trace-specific features for other traces as well. By contrast, although CGG also generates the 4 and 3 types that are equal to the real-world ones for HR and SN, respectively, the generated quantity has a large difference. Moreover, CGG also generates many call graphs that do not exist in the real-world ones.

6.2 The Effect of DGG on Evaluating Microservice Resource Scaling

DGG generates service dependency graphs that contain realistic characteristics from large-scale traces for producing benchmarks to assist in resource management studies. In this subsection, our goal is to validate that the benchmarks generated by DGG have a similar effect in investigating the resource efficiency of specific resource management strategies to the benchmarks directly constructed from real-world.

6.2.1 Microservice Scaling Strategies. From our trace analysis in Section 4, we identify two production characteristics of repeated calls and dynamic calling interfaces, which are required to be considered for microservice resource scaling. Based on these, we define the fine-grained call graph, where each vertex represents a microservice and its called interfaces, and each edge represents repeated calling times (Section 5.2.1). Using this fine-grained call graph, we propose the FineGrained-Scale strategy, which traces fine-grained call graph loads with Jaeger [17] at one-minute intervals, and allocates resources based on real-time monitored loads. For resource allocation decisions, we build offline linear regression models for each microservice in each call graph, with load as input and computing resource demand as output, as shown to be effective in recent studies [24, 38].

Moreover, prior microservice resource management works constructed coarse-grained call graphs [21, 24, 25, 31, 37, 38, 43], neglecting various interfaces and repeated callings, thus

Table 5: Experiment specifications

	Specifications
Hardware	Three nodes, Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz, 128GB Memory, 25 MiB L3 Cache
Software	Ubuntu 20.04.6 LTS with kernel 5.15.0-107-generic, Docker version 24.0.5, Kubernetes version v1.20.4, Golang version 1.19.3, gRPC version 1.29.1

different fine-grained call graphs are mapped to the same coarse-grained call graph. With coarse-grained call graphs, the aggressive [21, 24, 31, 43] or conservative [25, 38] strategies can only be utilized for resource scaling, which scale according to the maximum or minimum possible computing resource usage (named MAX-Scale and MIN-Scale), respectively. We also adopt the above two strategies in our evaluations. For MAX-Scale/Min-Scale, the computing resources of a microservice is allocated based on its coarse-grained total load and the linear model of FineGrained-Scale that has the maximum/minmum resource allocation value online.

6.2.2 Investigation Setup. We use two sets of benchmarks. The first is DGG-generated dependency graphs in Section 6.1, based on which we generate six benchmarks (named from DGG-0 to DGG-5). The second is to randomly select six dependency graphs for the six clustering sets from the remaining 20% services in Alibaba traces as stated in Section 4.1, and generate six Real-world benchmarks (named from Real-0 to Real-5). We implement the DGG's and Real-world benchmarks based on μBench [10], which is an assistant benchmark generation tool that creates executable microservice applications from input call graphs.

In terms of call graph loads, for each DGG's benchmark with n call graphs, we utilize the query per minute of the top-n accessed call graphs in the corresponding service clustering set in the trace. Moreover, for each real-world benchmark, we directly utilize its realistic query per minute in the trace. We evaluate each benchmark for one hour with the trace data and set the QoS of each benchmark to its 95%-ile latency under no computing resource constraints. Table 5 shows the hardware and software configurations.

6.2.3 Tail Latency and Resource Allocation. Figure 12 shows the 95%-ile latencies of the six types of simulated benchmarks generated by DGG and real-world benchmarks from traces with the three strategies. We can observe that the 95%-ile latency relationships among the three strategies on each type of DGG's benchmarks is similar to that on each type of real-world benchmark. In detail, both FineGrained-Scale and MAX-Scale ensure the QoS on DGG's or real benchmarks, while Min-Scale violates the QoS by 2.7X and 2.5X on average for DGG's and real-world benchmarks, respectively.

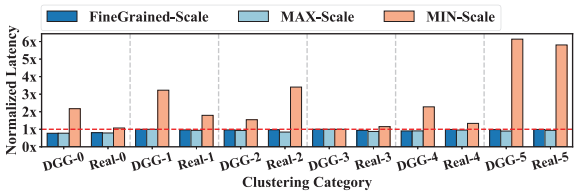


Figure 12: The 95%-ile latency of three strategies on DGG's and real benchmarks normalized to the QoS.

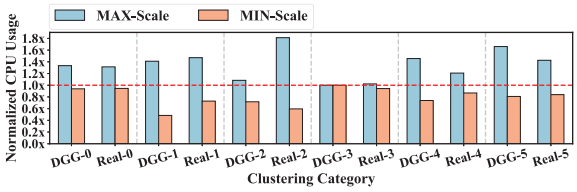


Figure 13: The CPU usage of MAX-Scale and MIN-Scale normalized to FineGrained-Scale on DGG's and real benchmarks, respectively.

Figure 13 also shows the total CPU core hour usage of the six types of DGG's benchmarks and real-world benchmarks with the three strategies. We can observe that the resource allocation relationships of the three strategies on DGG's and real-world benchmarks are also similar. For DGG's benchmarks, FineGrained-Scale has less resource usage than MAX-Scale, with an average reduction of 24.2% and a maximum reduction of 39.8%. For real-world benchmarks, FineGrained-Scale reduces the resource allocation by 27.3% on average and up to 44.8% compared to MAX-Scale. Moreover, the MIN-Scale has the minimum resource allocation on no matter DGG's or real-world benchmarks, but its QoS is violated.

The above results prove that the benchmarks generated by DGG can effectively reflect both the end-to-end performance and the resource scaling efficiency of different resource management strategies in real-world applications.

6.2.4 Summary of Benefits for Using DGG. Since DGG's generated benchmarks can well reflect the performance and resource efficiency of different resource management strategies for real-world applications, we summarize the benefits of using DGG on microservice resource scaling.

(1) Strategy Validation in Production. For the cloud, directly validating the efficiency of different resource management strategies on a large-scale online application may negatively impact on user experience and cost efficiency. In such cases, DGG can be used to generate small-scale benchmarks from online traces to explore the efficiency of various strategies, allowing cloud providers to select the most efficient one for production deployment.

(2) Microservice Scaling Studies for Research. Currently, the graph topology of benchmarks in academia is

quite simple, such as DeathStarBench [12], μSuit [42], and Online Boutique [13]. Using them to explore microservice resource management does not align with real-world scenarios. Therefore, researchers can utilize DGG to construct benchmarks from microservice traces for research, which better aligns with the characteristics of production microservices.

7 Conclusion

This paper proposes DGG for generating service dependency graphs of benchmarks that incorporate production-level features. Specifically, DGG uses a data handler to construct precise call graphs from the production traces and merges them into dependency graphs. It then clusters these dependency graphs into different categories based on their topological and invocation types. Based on the organized data, DGG uses a graph generator to generate service dependency graphs based on the random graph models that simulate real microservices invoking downstream microservices. Case studies show that DGG's generated dependency graphs closely resemble real traces. Moreover, for evaluating the same resource scaling strategy, DGG-generated benchmarks show the resource efficiency increase is 24.2% on average while real trace benchmarks show that value of 27.3%, proving that DGG can reflect similar results in evaluating microservice resource management strategies to real-world.

Acknowledgments

We sincerely thank our anonymous reviewers for their helpful comments and suggestions. This work is partially sponsored by the National Key Research and Development Program of China (2023YFB3001504), and National Natural Science Foundation of China (62232011, 62302302).

References

[1] Mohiuddin Ahmed, Raihan Seraj, and Syed Mohammed Shamsul Islam. 2020. The k-means Algorithm: A Comprehensive Survey and Performance Evaluation. *Electronics* 9, 8 (2020). doi:10.3390/electronics9081295

[2] Alibaba. 2024. cluster-trace-microservices-v2022. https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2022

[3] Nuha Alshuqayran, Nour Ali, and Roger Evans. 2016. A Systematic Mapping Study in Microservice Architecture. In *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*. 44–51. doi:10.1109/SOCA.2016.15

[4] 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 *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 533–546.

[5] Jop Briët and Peter Harremoës. 2009. Properties of classical and quantum Jensen-Shannon divergence. *Physical Review A—Atomic, Molecular, and Optical Physics* 79, 5 (2009), 052311.

[6] Liao Chen, Shutian Luo, Chenyu Lin, Zizhao Mo, Huanle Xu, Kejiang Ye, and Chengzhong Xu. 2024. Derm: SLA-aware Resource Management for Highly Dynamic Microservices. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 424–436. doi:10.1109/ISCA59077.2024.00039

[7] Yue Cheng, Zheng Chai, and Ali Anwar. 2018. Characterizing Co-located Datacenter Workloads: An Alibaba Case Study. In *Proceedings of the 9th Asia-Pacific Workshop on Systems* (Jeju Island, Republic of Korea) *(APSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 12, 3 pages. doi:10.1145/3265723.3265742

[8] 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 the 26th Symposium on Operating Systems Principles*. 153–167.

[9] Fabrizio Costa and Kurt De Grave. 2010. Fast neighborhood subgraph pairwise distance Kernel. In *Proceedings of the 27th International Conference on International Conference on Machine Learning* (Haifa, Israel) *(ICML'10)*. Omnipress, Madison, WI, USA, 255–262.

[10] Andrea Detti, Ludovico Funari, and Luca Petrucci. 2023. μBench: An Open-Source Factory of Benchmark Microservice Applications. *IEEE Transactions on Parallel and Distributed Systems* 34, 3 (2023), 968–980. doi:10.1109/TPDS.2023.3236447

[11] Kaihua Fu, Jiuchen Shi, Quan Chen, Ningxin Zheng, Wei Zhang, Deze Zeng, and Minyi Guo. 2022. QoS-aware irregular collaborative inference for improving throughput of DNN services. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 69, 14 pages.

[12] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 3–18. doi:10.1145/3297858.3304013

[13] Google. 2024. Online Boutique. https://github.com/GoogleCloudPlatform/microservices-demo

[14] Greg Hamerly and Charles Elkan. 2003. Learning the k in k-means. *Advances in neural information processing systems* 16 (2003).

[15] Darby Huye, Yuri Shkuro, and Raja R Sambasivan. 2023. Lifting the veil on {Meta's} microservice architecture: Analyses of topology and request workflows. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 419–432.

[16] Valeriu Manuel Ionescu. 2015. The analysis of the performance of RabbitMQ and ActiveMQ. In *2015 14th RoEduNet International Conference-Networking in Education and Research (RoEduNet NER)*. IEEE, 132–137.

[17] Jaeger. 2024. Jaeger: open source, distributed tracing platform. https://www.jaegertracing.io/

[18] Jeremy H. 2021. gRPC vs. REST: Key Similarities and Differences. https://blog.dreamfactory.com/grpc-vs-rest-how-does-grpc-compare-with-traditional-rest-apis

[19] Gopal Kakivaya, Lu Xun, Richard Hasha, Shegufta Bakht Ahsan, Todd Pfleiger, Rishi Sinha, Anurag Gupta, Mihail Tarta, Mark Fussell, Vipul Modi, Mansoor Mohsin, Ray Kong, Anmol Ahuja, Oana Platon, Alex Wun, Matthew Snider, Chacko Daniel, Dan Mastrian, Yang Li, Aprameya Rao, Vaishnav Kidambi, Randy Wang, Abhishek Ram, Sumukh Shivaprakash, Rajeet Nair, Alan Warwick, Bharat S. Narasimman, Meng Lin, Jeffrey Chen, Abhay Balkrishna Mhatre, Preetha Subbarayalu, Mert Coskun, and Indranil Gupta. 2018. Service fabric: a

distributed platform for building microservices in the cloud. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 33, 15 pages. doi:10.1145/3190508.3190546

[20] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing.

[21] Shutian Luo, Chenyu Lin, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, Huanle Xu, and Chengzhong Xu. 2024. Optimizing Resource Management for Shared Microservices: A Scalable System Design. *ACM Trans. Comput. Syst.* 42, 1–2, Article 1 (Feb. 2024), 28 pages. doi:10.1145/3631607

[22] 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 *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 412–426. doi:10.1145/3472883.3487003

[23] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, and Chengzhong Xu. 2022. An In-Depth Study of Microservice Call Graph and Runtime Performance. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 3901–3914. doi:10.1109/TPDS.2022.3174631

[24] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, Guodong Yang, and Chengzhong Xu. 2022. Erms: Efficient Resource Management for Shared Microservices with SLA Guarantees. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 62–77. doi:10.1145/3567955.3567964

[25] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu. 2022. The power of prediction: microservice auto scaling via workload learning. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco, California) *(SoCC '22)*. Association for Computing Machinery, New York, NY, USA, 355–369. doi:10.1145/3542929.3563477

[26] Martin Nally. 2020. REST vs RPC: What problems are you trying to solve with your APIs? https://cloud.google.com/blog/products/application-development/rest-vs-rpc-what-problems-are-you-trying-to-solve-with-your-apis

[27] memcached. 2024. memcached - a distributed memory object caching system. https://memcached.org/

[28] Meta. 2024. facebook distributed traces. https://github.com/facebookresearch/distributed_traces

[29] MongoDB. 2024. MongoDB: The Developer Data Platform. https://www.mongodb.com/

[30] nginx. 2024. nginx. https://nginx.org/en/

[31] 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 the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 46, 21 pages.

[32] 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 the third ACM symposium on cloud computing*. 1–13.

[33] Mohammad Reza Saleh Sedghpour, Aleksandra Obeso Duque, Xuejun Cai, Björn Skubic, Erik Elmroth, Cristian Klein, and Johan Tordsson. 2023. Hydragen: A microservice benchmark generator. In *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*. IEEE, 189–200.

[34] Vishwanath Seshagiri, Darby Huye, Lan Liu, Avani Wildani, and Raja R Sambasivan. 2022. [SoK] Identifying Mismatches Between Microservice Testbeds and Industrial Perceptions of Microservices. *Journal of Systems Research* 2, 1 (2022).

[35] Jiuchen Shi, Kaihua Fu, Quan Chen, Changpeng Yang, Pengfei Huang, Mosong Zhou, Jieru Zhao, Chen Chen, and Minyi Guo. 2022. Characterizing and orchestrating VM reservation in geo-distributed clouds to improve the resource efficiency. In *Proceedings of the 13th Symposium on Cloud Computing*. 94–109.

[36] Jiuchen Shi, Kaihua Fu, Jiawen Wang, Quan Chen, Deze Zeng, and Minyi Guo. 2024. Adaptive QoS-Aware Microservice Deployment With Excessive Loads via Intra- and Inter-Datacenter Scheduling. *IEEE Trans. Parallel Distrib. Syst.* 35, 9 (Sept. 2024), 1565–1582. doi:10.1109/TPDS.2024.3425931

[37] Jiuchen Shi, Jiawen Wang, Kaihua Fu, Quan Chen, Deze Zeng, and Minyi Guo. 2022. QoS-awareness of Microservices with Excessive Loads via Inter-Datacenter Scheduling. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 324–334. doi:10.1109/IPDPS53621.2022.00039

[38] Jiuchen Shi, Hang Zhang, Zhixin Tong, Quan Chen, Kaihua Fu, and Minyi Guo. 2023. Nodens: Enabling Resource Efficient and Fast {QoS} Recovery of Dynamic Microservice Applications in Datacenters. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 403–417.

[39] Jiu-Chen Shi, Xiao-Qing Cai, Wen-Li Zheng, Quan Chen, De-Ze Zeng, Tatsuhiro Tsuchiya, and Min-Yi Guo. 2022. Reliability and incentive of performance assessment for decentralized clouds. *Journal of Computer Science and Technology* 37, 5 (2022), 1176–1199.

[40] Giannis Siglidis, Giannis Nikolentzos, Stratis Limnios, Christos Giatsidis, Konstantinos Skianis, and Michalis Vazirgiannis. 2020. GraKeL: A Graph Kernel Library in Python. *Journal of Machine Learning Research* 21, 54 (2020), 1–5.

[41] Raj Srinivasan. 1995. *RPC: Remote procedure call protocol specification version 2*. Technical Report.

[42] Akshitha Sriraman and Thomas F. Wenisch. 2018. μSuite: A Benchmark Suite for Microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. 1–12. doi:10.1109/IISWC.2018.8573515

[43] Lalith Suresh, Peter Bodik, Ishai Menache, Marco Canini, and Florin Ciucu. 2017. Distributed resource management across process boundaries. In *Proceedings of the 2017 Symposium on Cloud Computing*. 611–623.

[44] Huangshi Tian, Yunchuan Zheng, and Wei Wang. 2019. Characterizing and synthesizing task dependencies of data-parallel jobs in alibaba cloud. In *Proceedings of the ACM Symposium on Cloud Computing*. 139–151.

[45] Takanori Ueda, Takuya Nakaike, and Moriyoshi Ohara. 2016. Workload characterization for microservices. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. 1–10. doi:10.1109/IISWC.2016.7581269

[46] Maarten Van Steen and A Tanenbaum. 2002. Distributed systems principles and paradigms. *Network* 2, 28 (2002), 1.

[47] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. 2019. Microservice based architecture: Towards high-availability for stateful applications with kubernetes. In *2019 IEEE 19th international conference on software quality, reliability and security (QRS)*. IEEE, 176–185.

[48] Yingying Wen, Guanjie Cheng, Shuiguang Deng, and Jianwei Yin. 2022. Characterizing and synthesizing the workflow structure of microservices in ByteDance Cloud. *Journal of Software: Evolution and Process* 34, 8 (2022), e2467.

[49] 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 the 26th ACM*

international conference on architectural support for programming languages and operating systems. 167–181.

[50] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. 2018. Benchmarking microservice systems for software engineering research. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 323–324. doi:10.1145/3183440.3194991