Received: 20 October 2021 | Revised: 23 December 2021 | Accepted: 9 May 2022

DOI: 10.1002/smr.2467

RESEARCH ARTICLE - EMPIRICAL

Journal of Software: Evolution and Process WILEY

Characterizing and synthesizing the workflow structure of microservices in ByteDance Cloud

Yingying Wen | Guanjie Cheng | Shuiguang Deng | Jianwei Yin

Computer Science and Technology, Zhejiang University, Hangzhou, China

Correspondence
Yingying Wen, Computer Science and Technology, Zhejiang University, Hangzhou, China.
Email: 11821102@zju.edu.cn

Funding information
National Natural Science Foundation of China, Grant/Award Numbers: 62125206, U20A20173

Abstract

Modern Cloud applications have evolved from monolithic systems to numerous distributed microservices whose workflows interact via Remote Procedure Calls (RPCs). The benchmarks focusing on individual microservice components are insufficient because the reference relationships between components, which form the microservice workflow structure, are another critical aspect of microservice applications. Unfortunately, understanding the characteristics of microservice workflow in the production Cloud remains a missing piece in the literature, which prevents the representative of microservice benchmarks. In this paper, we fill this gap by characterizing and synthesizing the microservice workflows based on the trace data of the Toutiao application that is running on ByteDance Cloud. We examine the microservice workflows starting from DAG graphs, introduce observed properties that are easy to ignore but important, show the artificiality of the workflow by statistic description, and explore the high cost of network overhead. We further synthesize the workflow following the characteristics observed. The extensive evaluations show that the synthesized microservice workflows have consistent statistical characteristics as the production ones. A case study applying the synthesized workflows proves its usability.

KEYWORDS
ByteDance Cloud, microservice architecture, production data characterization, trace data analysis

1 | INTRODUCTION

Cloud computing has caused a revolution in our way of developing and deploying software services. Attracted by the ability of microservice architecture to independently update and redeploy the code base of individual microservices, increasing applications' scalability, portability, updatability, and availability, several companies such as Amazon[1] and Netflix[2] have adopted microservice architectures. These microservice components are composed via standardized Remote Procedure Calls (RPC) interfaces, such as Google's gRPC[3] or Facebook/Apach's Thrift.[4] The dependencies among components and their communication cost, once insignificant for monolithic applications, can now dominate the microservice regime.[5,6]

Microservice deployment/scheduling is a representative research problem in Cloud that needs to be workflow-aware. To deploy the microservice components at the best possible machines with minimum inter-machine traffic, the microservice workflow is modeled as a dependency graph supporting the K-cut algorithm.[7] To optimize resource cost with deadline constraints, the elastic scheduling method[8] prioritizes running resources to the tasks in the critical path of microservice workflow. For balancing the resource usage with consideration of user requests and thus

reducing the energy consumption, the routes of user requests are modeled as a critical aspect.[9] All these optimization problems are workflow-dependent.

Benchmarks is one of the basic support elements for exploring the most effective optimization strategies. Prior studies constructed the microservice benchmark with a single workflow formed by the static function calling references between components, such as the DeathStarBench suite[10] that includes six end-to-end services and the μSuite[11] that comprises four OLDI services to form their corresponding static workflows. But simulating several application workflows is far away from characterizing the heterogeneous microservice workflows of real applications running in the production Cloud.[12] The inadequate representative of benchmarks on the batch jobs is shown by the study of characterizing the Alibaba trace data.[13] But characterizing the microservice workflow remains a missing piece in the literature.

In this paper, we present a panoramic view of production microservice workflow structure through an in-depth characterization study on the ByteDance trace data. The trace data is collected by the RPC-layer-based tracing function[14,15] to profile the remote calls among microservice components. Our work includes two parts, the characterizing and synthesizing study.

First, we characterize the workflow structure by modeling the trace logs. We find essential properties of the real-world applications' workflows that have not been emphasized before. The artificiality of the workflow structure is described from statistic analysis on the number of nodes in the trace tree, the depth of the trace tree, and the number of remote calls started by a parent node. The long-duration trace does not need to be large, whereas large trace tends to run longer; and the small number of workflows with large size consume a non-negligible proportion of the CPU resource, which are worth attention. We also explore the overhead of the network that is impacted by the message size and the basic connection establishing phase.

Second, we synthesize the workflow structure starting from the trace tree generation. The generation algorithm is based on the probability model, and a latent factor, the depth layer of the parent node, is considered. We then separate the remote calls, following the properties we observed, into synchronous and asynchronous calls. Runtime costs, including the running time of a component and the message size, are generated. We show the similarity between generated and real ones from trace shape and trace duration dimensions. A case study is conducted to prove its usability by applying the generated microservice workflow to find a better component placement schema.

We summarize our key contributions as follows.

- We conduct a comprehensive analysis of the properties of the microservice workflow structure and critical characteristics at scale.
- We synthesize the microservice workflow topology and shows the consistency of the generated data with the production data.
- We discuss the possible performance optimization opportunity about the microservice placement by considering the workflow structure.

We organize this paper as follows. Section 2 introduces the related work. Section 3 describes the trace data we use. Section 4 characterizes the trace data by introducing the properties, the artificiality of workflows, and network overhead. The synthesizing method and validation are shown in section 5. We discuss the implications to microservice deployment and limitations of analysis in Section 6. Section 7 concludes our work.

2 | RELATED WORK

This section introduces related work from three areas, including microservice benchmarks and characterization, the microservice performance optimization, and works on characterizing the production data.

Microservice benchmarks and characterization. The existing microservice benchmarks are designed for simulating the microservice architecture with workflows formed by the function calling references among components. DeathStarBench suite[10] includes six end-to-end services designed for different service scenarios. Each service forms its own workflow structure without varying. And the μSuite[11] that comprises four OLDI services to make its components' workloads more typical. Pattern based approach[16] also only focused on single microservices. Prior characterization studies[17–19] on microservices mainly focus on the resource usage aspect of individual microservices and comparing the performance of microservice applications with monolithic applications to show the characteristics. Full microservice applications and dependencies across microservices have not been supported, which motivates us to fill the gap to characterize the dependencies among microservices.

Microservice performance optimization. Performance optimization is a long-standing challenge. Predictable performance is even more critical as Cloud services transition to microservices, as the hotspots propagate and amplify across dependent services. Seer[20] is designed to identify application-level design bugs. MicroRCA[21] is designed to find the root cause. MicroService Capacity (MSC)[22] models the relationship between service level objectives and resources available for microservices. Workflow scheduling mainly focuses on the resource offering areas to meet service objectives.[23–25] The network overhead caused by RPC is acknowledged. SmartGridRPC[26] intends to match the RPC workflow for the grid network environment to reduce the RPC cost. μTune[6] designs auto-tuned threading method to interact with RPC interfaces. As pointed by Jayasinghe's analysis,[27] the number of service calls that impact the performance has not been considered, which also need the support of workflow characterization.

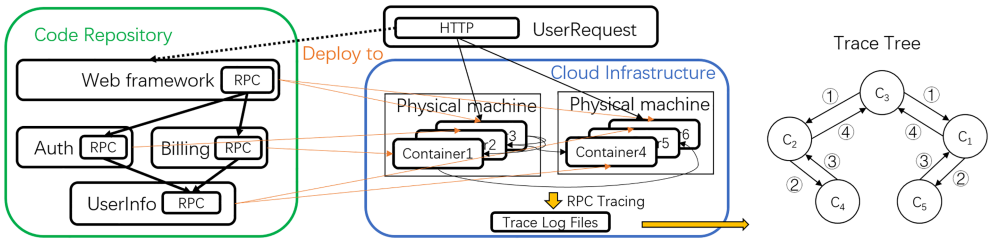


FIGURE 1 The architecture of trace data collection, including the container-based running of microservice applications, container-based running, and RPC-layer-based tracing

Characterizing the production data. For helping to understand the characteristics of production environments and motivate research to real application scenarios, many pieces of literature describe the production data. Huangshi et al[13] utilizes Alibaba's open-source trace data to describe the task dependencies of Data-Parallel jobs. Alibaba's trace data is also used to characterize the workload co-location and resource usage condition.[28–30] Google uses its system-wide profiling data[31] to characterize Cloud usage[32] and shows its heterogeneity and dynamicity.[33] We are the first one utilizing the trace data to characterize the microservice workflow structure.

3 | OVERVIEW OF THE TRACE DATA

In this section, we overview the trace data, focusing on the information related to the microservice workflow. We show the architecture of data collection in Figure 1. There are three key parts: first is the container-based running of microservice applications; second is the RPC-layer-based tracing; third is the parsing of trace log files to trace trees.

3.1 | Container-based deployment

Container-based virtualization offers operating system level abstraction and isolation to allow a physical machine to be shared among multiple containers.[34] The static source code repositories of the microservice components need to be run up with running resources. The containers are the dynamic running instances that manage the resources and run the source code up. For the diverse resource requirements of different components, the microservice architecture supports the independently scaling up of each component. For a component, there is more than one container instance in Cloud for running it. The number of containers for a component and the amount of resource in a container can both be auto-scaled.[35]

Except for the resource management supported by containers, the requests to a component would be distributed to its running instances (containers) with load balance consideration,[36] which makes users unaware of the infrastructure management.

3.2 | RPC-layer-based tracing

To efficiently develop and manage communication between components, the implementation of microservice components usually utilize an RPC framework, like the eRPC[37] and gRPC framework,[3] to unify the RPC communication protocols. In ByteDance Cloud, our RPC framework is named as kitex,[*] which implements the tracing technique that is similar to the OpenTracing API.[†] The traced results are the tracing logs stored in a distributed database to support future analysis.

We show the implementation details of the tracing function in Figure 2. In order to trace across container boundaries in distributed systems, the tracing function needs to be able to continue the trace. For example, in the RPC framework design, the client and server structure is the basic unit for crossing the container boundary. Thus a trace context is injected into a client request carrier, and server-side extracts the trace context from this carrier to continue a trace. When a server extracts the trace context successfully, it would recognize that it is a child component of an existing trace. In contrast, when the server fails to exact the trace context, it would treat itself as a root component of a trace and create a trace context to denote a new trace.

[*]https://github.com/cloudwego/kitex.
[†]https://github.com/opentracing.

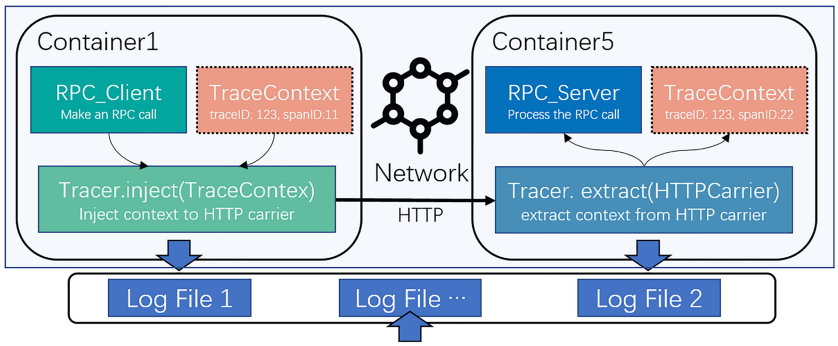


FIGURE 2 The implementation of the tracing function based on the RPC framework by propagating the trace context to identify a trace

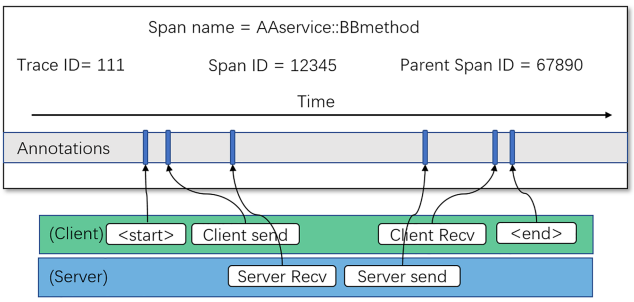


FIGURE 3 A detailed view of span with information from two hosts

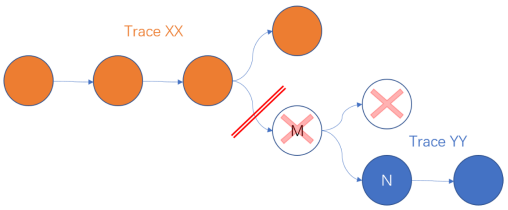
TABLE 1 An example to show the trace log information

Trace ID	Span ID	Parent span ID	Component name	Type (client/server)	Start time	Duration
109826AF	439975457609203553	0	user.base	Server	0 ms	5.18 ms
109826AF	439975457609203554	439975457609203553	user.base	Client	0.27 ms	4.76 ms
109826AF	8018155997965191177	439975457609203554	punish_center.status	Server	2.63 ms	0.52 ms

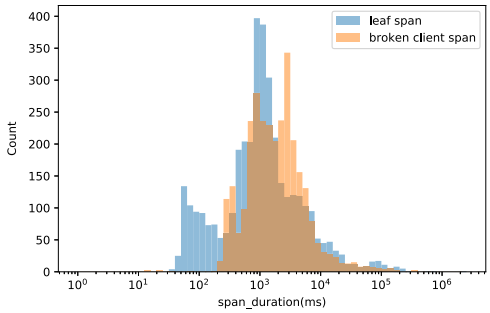
3.3 | Parsing the trace log files

The key challenge to parse the log file is to combine the information across the container boundaries. We name the process of a remote call, as shown in Figure 2, as a ***span***. It is important to note that a span can contain information from multiple hosts of both the client and server processes, as the example shown in Figure 3. A span belongs to a trace: A trace is identified by a trace ID; A span contains two span-IDs that identify the client and server processes. And the span would also record its parent span ID to denote the span that starts it, which helps form a trace calling tree.

Here is an example to illustrate the log files. Table 1 is the information of a trace with two components, "user.base" and "punish_center.status," and a remote call between them. There are two calls across the boundaries: the call started from the user interface to call the "user.base" components, and the call started by "user.base" to call the "punish_center.status." As we do not have the trace logs of the user interfaces (like the mobile Apps and websites), we only trace the remote calls between components, which makes the root span of a trace start from a server type record of the component in the Cloud. But there are still two references in the table: one is span ID "439975457609203554" to "439975457609203553"; another is span ID "8018155997965191177" to "439975457609203554." This is because, following the user request to "user.base" that is run up as a server, "user.base" starts a remote call and create a client process record in the same container and refer to the former server process as parent span id for not losing the structure information. Thus, only the reference of "8018155997965191177" to "439975457609203554" represents the remote call. The reference of "439975457609203554" to "439975457609203553" denotes the client process started at the same container as its parent server process running at.



(A) Illustration on the cause of dirty data.



(B) The difference of span duration between broken spans in the user started traces and the leaf spans in all traces are similar.

FIGURE 4 Data cleaning

3.4 | Data cleaning

Though the RPC-layer-based tracing already covers most of the components that base on the RPC framework, there are still some of components that have not instrumented the tracing function, which cause the dirty data. For accurately understanding the characteristics of microservice workflow, we further clean the collected data.

About the cause of dirty data, specifically, for a microservice workflow as shown in Figure 4A, once there is an inner node M that does not support the tracing function, the trace tree would be cut into pieces. This is because the trace context indicating the trace XX would fail to spread to the sub-tree starting from node M. When there is a node N in the sub-tree that has the tracing function and there is no trace context spread to it, node N would create a new trace context that denotes a new trace YY. But in fact, trace YY and trace XX are both parts of a single workflow. This condition makes the data easy to be dirty. For example, a workflow crosses 100 components and the coverage rate is 99%. Though there is only one component that lacks the tracing function, which even not in the critical path, but from the trace record view, it introduces 50% trace records impacting the statistics of the trace data.

We, therefore, clean the trace records to reduce the impact of trivial records. Based on the architecture of data collection, we find the requests from users are directed first to the service mesh and labeled as call from service mesh layer, like the NGINX,[‡] whereas the RPC request from remote components are labeled as call from RPC layer. Thus we use the request type of root span to classify the trace records into user started traces (such as the Trace XX in Figure 4A) and broken sub-traces (such as the Trace YY in Figure 4A). We find the time granularity of these broken spans is not larger than the time granularity of leaf spans as shown in Figure 4B. This means that the broken spans losing its sub-traces in user started traces is close to the leaf spans. Broken spans in user started traces have little impact on the running characteristics when filter out the sub-trace records and would not impact our understanding of the current workflow topology.

3.5 | Data volume

The coverage of tracing function on service modules already reaches 78% in our Cloud and covers critical running paths. The coverage rate is high enough to help us recognize the service workflow structure and characterize the production environment. We choose the Toutiao application as our primary analysis target, whose microservice modules have the highest tracing coverage rate. Though its business domain is showing news, the modules involved in our analysis reach tens of thousands, such as the functions of recommendation, advertising, e-commerce, short-videos, and so on. The data for analysis includes 128,290,337 traces, 17,332 types of root spans coming from 3,388 microservices, and RPC calls 36,260 function interfaces coming from 16,284 microservices.

4 | WORKFLOW CHARACTERIZATION

In this section, we describe the characteristics of microservice workflow. We first clarify the DAG graph model and the properties of microservice workflow. Next, based on the DAG graph, we describe the workflow structure from shape size, tree depth, out degrees of nodes to show the

[‡]https://www.nginx.com/.

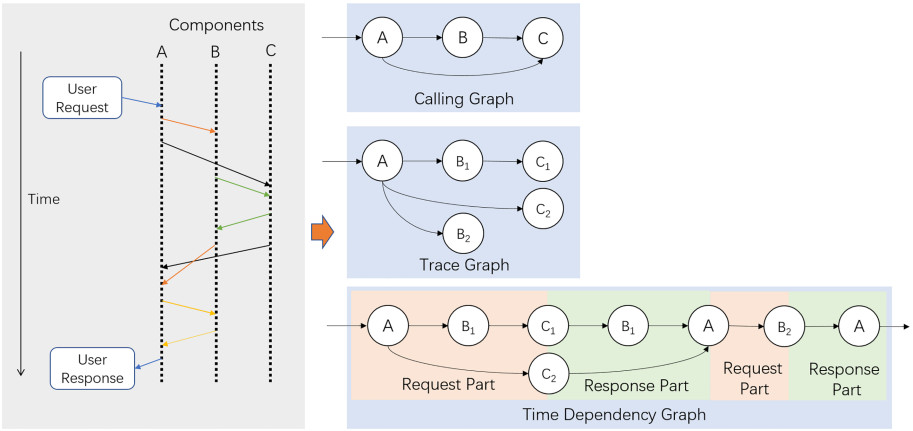


FIGURE 5 Three types of DAG graphs to model the microservice workflow

artificiality of microservice workflow. We further explore their correlations with trace duration and CPU usage conditions. The high network overhead is also observed to find its relationship to message size dimension.

4.1 | DAG graphs

Workflows are usually denoted as DAG graphs. The links between nodes denote the calling relationships among microservices. But the calling relationship is not enough to describe the workflow structure. Thus we introduce the trace tree graph whose links between nodes denote the dependencies of requests and responses. We show an example to compare three types of DAG graphs in Figure 5 to show their differences.

- **Calling Graph** describes the calling relationships among components. A calls B and C, whereas B calls C sometimes. The calling graph shows the static dependencies formed by the source code. But from the dynamic running view, the thread of the C component running for B is independent of the thread of the C component running for A, which makes the calling graph fail to describe the running structure.
- **Trace Graph** compensates this defect by constructing itself from trace logs to reflect the running dependency, which would treat independent running threads as different nodes in this graph, as shown by the B_1, B_2, C_1, and C_2. We would call the node in the trace tree as **module** in this paper to differentiate the static component. But trace graph fails to reflect the time dependency, such as the condition that B_2 starts running after the response of B_1.
- **Time Dependency Graph** reflects the time dependencies constructed directly from the left part, timeline illustration. Though the trace graph can reflect the running structure, it lacks the ability to model the workflow from the time dimension. Thus we introduce a time dependency model to help with understanding the trace graph.

4.2 | Observed properties of workflows

In this section, observed from the trace data, we find some properties that have formerly been ignored that would impact the design of microservice benchmarks extremely. For the current design of microservice benchmarks, like the DeathStarBench, user requests of the same service have the same workflow; all the remote calls started by a parent node would be treated as executing in parallel; all the modules of the service would cleanly appear once for each workflow. But in fact, there are some properties need to be considered for future benchmark designs:

The workflows of a single service would possibly be diverse across user requests. We find, though the calling graph of a specific service keeps still (without the change of implementation logic), the trace tree of this service would be diverse across user requests because the impact of diverse parameters. For showing the diversity, we bin the traces by the function interfaces of their root spans (we call them the entering function interfaces). For each entering function interface, we show the distribution of the number of nodes in their trace trees. There is an example result that contains 32 function interfaces shown in Figure 6. We can find that the number of nodes in trace trees of a single service is not the same, which means the workflow structures are different even for the same service.

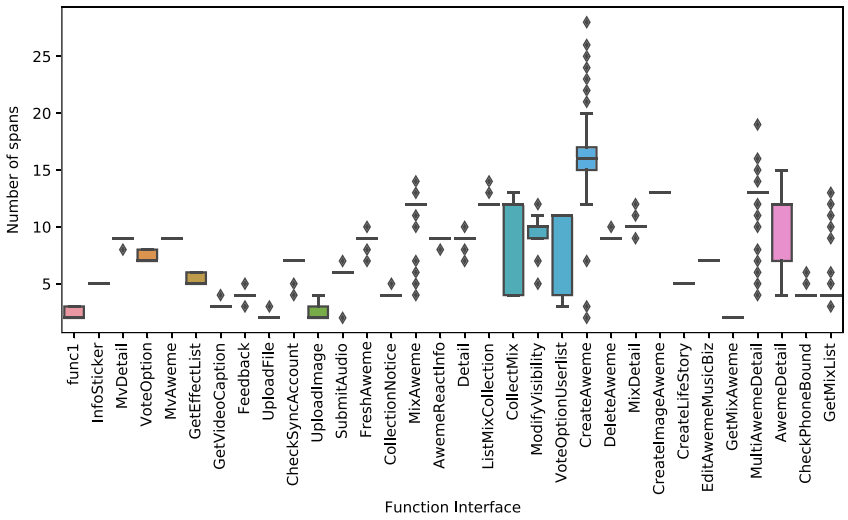


FIGURE 6 An example to show the number of spans in a trace started by the same function interface varies a lot

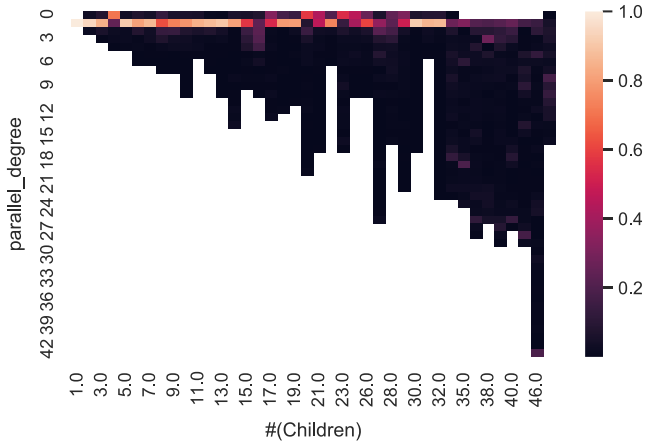


FIGURE 7 The parallel degree under different numbers of children

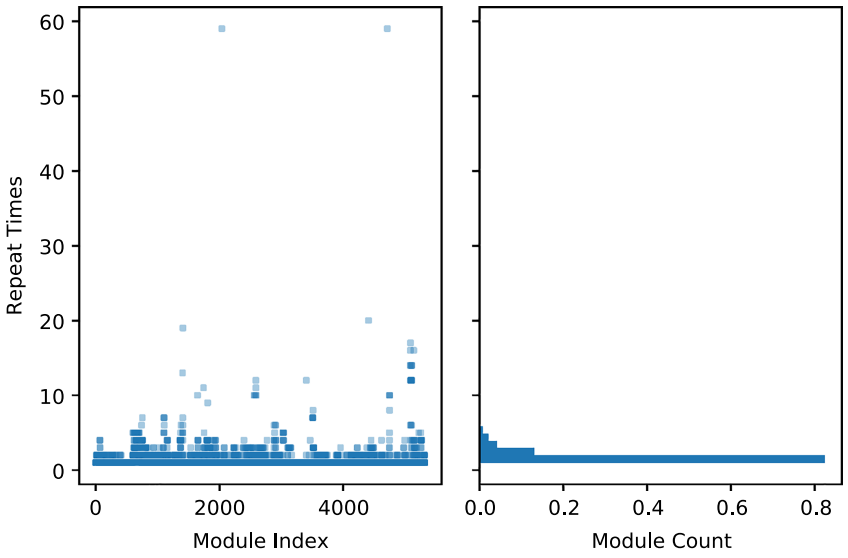


FIGURE 8 The distribution of the repeat times of modules called by a parent span

The remote calls started by a parent node are not executed independently and parallelly. Except for the call structure impacting the running time, the synchronization/asynchronization between calls is another critical dimension that needs exploration. When two calls are synchronous, one call must wait for the return of the other call. And from the time dimension, the running of these two calls is serial that one runs first and the second follows, such as the B_2 and B_1 in Figure 5. When two calls are asynchronous, one call is independent of the other call. Thus it is possible that the running of these two calls is in parallels, such as the C_1 and B_1 in Figure 5. When a module starts several remote calls, these calls can be synchronous or asynchronous.

From the trace data, there is no direct information to know if two calls are synchronous or not. Hence, we define a parallel degree value to denote the number of calls run parallelly at a time point. If all the remote calls of a parent node run independently and parallelly, the parallel degree value would approach the number of remote calls for most of the running time. But shown by Figure 7 (The color means the time proportion), we can find that the parallel degree is 2 in most of the running time, and the highest parallel degree is less than the number of remote calls. This means remote calls run both synchronously and asynchronously. It is illogical to treat all remote calls of a parent node as parallel running units.

Some components would be invoked repeatedly in a workflow to satisfy a user request. When we limit our observations to the traces with more than one span, we find module recurrence in these traces is not rare that module recurrence happens in 89.37% of these traces. But this does not mean most of the modules in the trace would be called repeatedly. As shown in Figure 8, about 80% of these modules are only called once by its parent span. The repeat times of 98.16% samples are shorter than 5. And the extremely large repeat time is not typical behavior.

4.3 | The artificiality of workflow structure

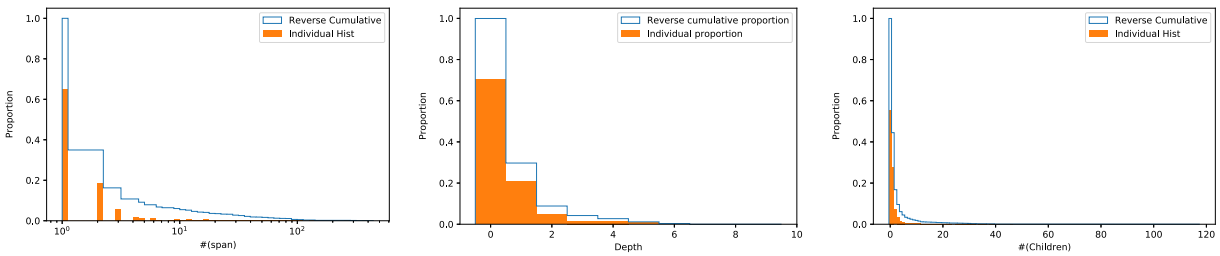
The dimensions we use to describe the artificiality of workflow structure include the number of nodes in the trace tree, the depth of the trace tree, and the out degrees of the nodes in the trace tree. We also explore the correlations between performance and workflow structure to show the importance of small proportion traces with large shapes.

4.3.1 | Statistic description and analysis

The size of the trace follows a long-tail distribution that most of the trace records contain a small number of nodes, and several trace records contain a huge number of nodes. We show the number of nodes in a trace in Figure 9A. We find: (1) About 65.04% of the trace records only have one span. This means lots of user requests can be satisfied by the web framework interface layer rather than call for help from other remote components. (2) The proportion of traces whose number of spans is less than 5 reaches 90.84%, and the proportion of traces whose number of spans is less than 13 reaches 95%. The maximum size reaches 391 spans of a trace. When benchmarking the microservices, the number of components involved by a user request can be limited to a smaller reasonable interval.

It is still a long-tail distribution of the depth that most of the tree structures are shallow, and a few trees are deep. As shown in Figure 9B, traces with 0 depth are the ones that only have a single span (65.04%). The proportion of traces with 1 depth reaches 26.06%, whereas the total proportion of the traces with more than one span is 34.96%. And the maximum depth is 10. The calling trees with at most 4 or 5 depths cover most conditions (99%).

The number of remote calls started by a parent node varies a lot, but the majority are limited to several possible values. The distribution of the number of children is shown in Figure 9C. 55.46% of recorded spans have no child. The proportion of spans with less than 3 children is 90.36%, and those with less than 5 children are 95.46%. The maximum number of children that a span has is 118. But this large number of remote calls is caused by module recurrence rather than a function that depends on many remote modules.



(A) Number of Spans (B) Depth of Trace (C) Number of Children

FIGURE 9 The distribution of three dimensions to show the trace shape

These statistic descriptions form a basic acknowledgment of the rough sizes and shapes of workflows. We would further utilize these statistic distributions to help with workflow synthesis in later sections.

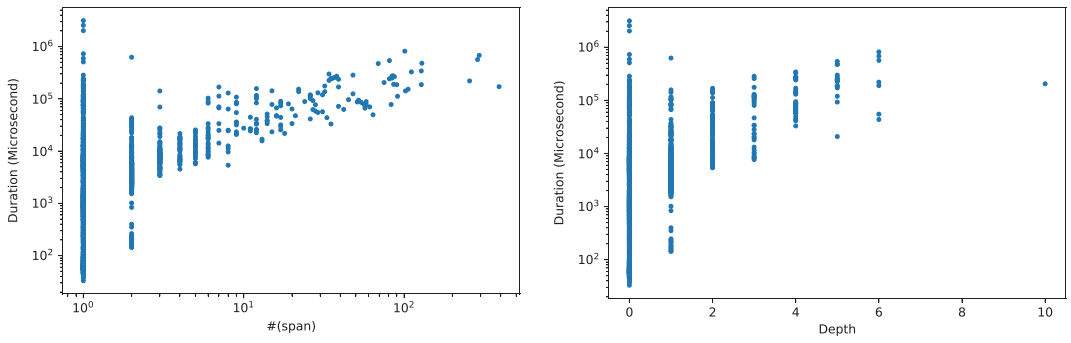
4.3.2 | The correlations between performance and workflow structure

We describe the performance from two major aspects, the running time of a workflow and the resource cost.

Running time performance. We use trace duration to show the running time performance, which means the time from the user request sent to the response received. We explore the correlation between the trace duration and the number of spans shown in Figure 10A and the correlation between the trace duration and the depth of the trace tree shown in Figure 10B. With the linear increase of the x-dimensions (the number of spans and depth), the lower bound of the trace duration increases exponentially (the y-axis is in log scale). The maximum duration of a trace with 2 spans and 1 depth is 625977 microseconds. The minimum duration of a trace with more than 2 spans is 3418 microseconds. And the minimum duration of a trace with a depth larger than 1 is 5362 microseconds. **We can find that long-duration traces do not need to be large in shape, but large shape traces would run for a long time**.

CPU usage performance. The former description from the structure view shows that the majority of traces are small in size. But from the CPU usage view, we find that the minority traces with large size consume a non-negligible proportion of the CPU resource. Because trace data does not contain the resource usage information, we combine other profiling data containing CPU usage from module granularity. Despite the lack of more detailed information to deduce the accurate CPU usage for every trace, we can still gain some knowledge from current data. We separate the traces according to their number of spans (larger than 10 or not) into two classes: the large and small traces. The modules involved by these traces hence can be separated into three sets: (1) The modules that only appear in small traces, (2) The module that only appears in large traces, and (3) the modules that appear in both classes of traces.

When we intend to separate the traces into two parts with 5% and 95% proportion, we get the classification critical as the number of spans is more than 10 or not. The statistical results are shown in Table 2. We can find the large traces, whose proportion is only 5.43%, contains more than 49.55% modules and consumes more than 33.67% resource. The resource here means the CPU quota. If we roughly estimate the result by allocating the "Both" part proportionally into "Small" and "Large" parts, we can get a rough result to show that 5.43% large traces contains $\frac{49.55}{31.12+49.55} \cdot 19.34\% + 49.55\% = 60.24\%$ modules and consumes $\frac{33.67}{6.54+33.67} \cdot 59.79\% + 33.67\% = 83.73\%$ CPU resource. **We can find that, though the traces with large size are the minority, it is necessary to pay attention to them when optimizing performance**.



(A) Spans → Trace Duration (B) Depth → Trace Duration

FIGURE 10 The relationship between dimensions

TABLE 2 The small number of traces contains a large proportion of modules and consume non-negligible CPU resource

Traces			
	Small trace		**Large trace**
Number of traces	94.57%		5.43%
Components			
	Small only	**Both**	**Large only**
Number of components	31.12%	19.34%	49.55%
CPU usage proportion	6.54%	59.79%	33.67%

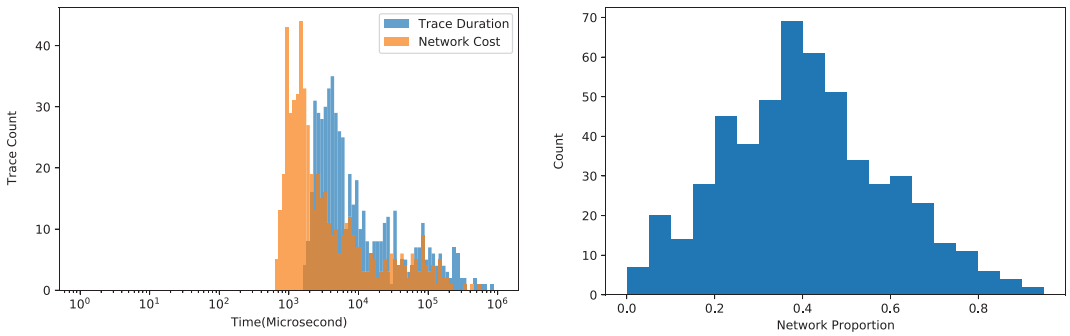
4.4 | The overhead of network communication

Prior works[5,6] report that the network cost means a lot to the microservice architecture. For validating this claim, we estimate the overhead of remote calls. The network overhead can be deduced from the client and server annotations in the span states. We name the difference between client start time and server start time as the sending cost and the difference between client end time and server end time as the receiving cost. Some of the user requests can be satisfied by the web framework layer without calling other modules. Hence, we only contain the traces that have RPC costs in the following analysis.

The distribution of trace duration (service running time) and network cost is shown in Figure 11A, and the distribution of the proportion of network cost to trace duration is shown in Figure 11B. The network cost here is the sum of all network cost in a trace. The average proportion of network cost is 44.09%. This supports the claim of prior works that the network cost in the microservice architecture is high. We further explore the impact factors.

We find that the message size of the communication is one of the impact factors. In Figure 12A, we show the relationship between message size and network communication cost from sending cost and receive cost, respectively. The network cost here is the time cost for each remote call. With the growth of message size, the network cost becomes higher (We need to notice the scale of the axis is log10, which makes the increasing trend flatten). When the message size is small, like smaller than 10^3, the network cost would not be impacted by message size and scatter around 1ms. But when the message size becomes larger, the impact of message size becomes dominant, and the network cost increases with message size. Moreover, the network performance differs little between request (send) and response (receive) phases. And the impact of message size is similar for these two phases.

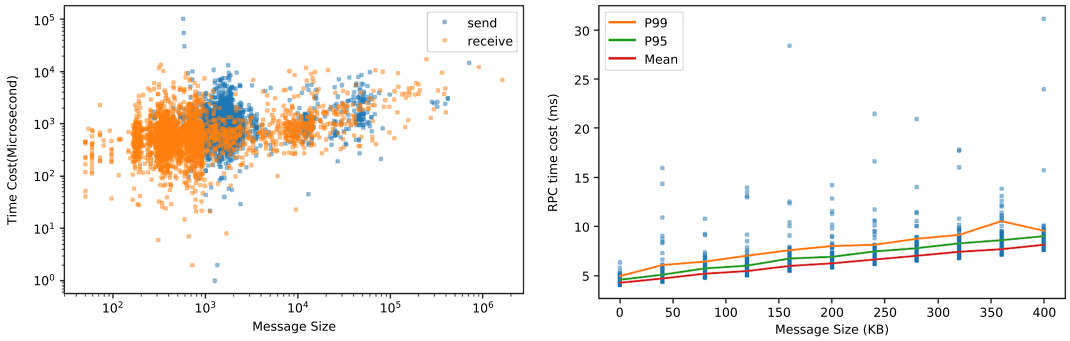
From Figure 12A, **we deduce there may be a basic cost of connection establish that would not be impacted by the message size**. But caused by the heterogeneity environment, it is not clear enough to prove the deduction. Thus, we establish an experiment to show the existence of a connection establishing phase between two components with a clearer view as shown in Figure 12B. In our experiment, the type of message is a string with a specified string length to reduce the impact of the environment. We find, when the message size is 0, there is a basic remote call cost. This experiment result also shows a clearer relationship between message size and time cost.



(A) The distribution of trace duration and its network cost.

(B) The distribution of the proportion of network cost.

FIGURE 11 The distribution and proportion of network cost



(A) Statistic results from the send and receive aspects respectively

(B) The experiment result

FIGURE 12 The relationship between message size and network cost

5 | WORKFLOW SYNTHESIS

For synthesizing the workflow, there are four major elements considered in this section. We first utilize the hierarchy probability model to generate trace trees according to the formerly introduced characteristics and keep the same characteristics in the synthesized results. Second, we explore the proportions of synchronous calls to form the structure of the time dependency graph. We further add the runtime dimensions to the generated graph, including the duration time of components and the communication cost between them. Finally, we validate our generated traces by comparing the distribution of trace duration to the real-world ones. We also show a case study applying our synthesized workflow to explore the effect of microservice placement.

5.1 | Trace tree shape generation

We synthesize the traces starting from the number of remote components called by a parent one. The number of called components by a parent may be impacted by many latent factors, such as the workload type of the parent spans, the depth layers in the traces of the parent spans, the user request contents, and the load levels. With more latent factors introduced into the model, this model would be more accurate and more complicated. In this paper, we focus on one of these latent factors, the depth layer of the parent component in the trace tree, to generate the traces. We regard the root node of a trace tree as in the 0^{th} depth layer. The called component by a parent in i^{th} depth layer is regarded as in the $(i+1)^{th}$ depth layer.

To explore the depth layer's impact, we show the distributions of the number of children for nodes binned to different depth layers. As shown in Figure 13, we can find the distribution of the number of calls by a component differs with the depth layer. The distributions of the first 5 layers from the 0^{th} depth layer to 4^{th} depth layer are similar, and they are different from the deeper layers. Deeper layers have a smaller number of children when compared with shallow layers.

Generation algorithm. We denote the distribution of the number of calls in d^{th} depth layer as P_d that contains the probabilities of all possible numbers of the calls denoted as $\{P_d(c_d = k) = p_{dk}, 0 \le k \le K_d\}$ where K_d means the maximum number of calls in d^{th} depth. When given depth information, d of a span, the probability of the number of children would follow the distribution of P_d. We can generate the number of children according to these distributions. With the input of $\{P_d, 0 \le d \le MaxDepth\}$ and the generation method shown in Algorithm 1, we can get the output of a synthesized trace. We generate the traces starting from the root span that is in the 0^{th} depth layer. According to the corresponding distribution, the number of children can be settled. Then for each child with depth increased 1, the generation iterates until every end-node has no child. We denote the output tree by the pairs of node id and parent node id. Because the last layer spans (deepest spans) have no child and the probability of no child for a span would make the calling process stop and finish the trace construction, the synthesizing process would converge.

Algorithm 1 Generate traces

Require: $\{P_d, 0 \le d \le MaxDepth\}$: The distribution of the number of children in d^{th} depth
Ensure: T: the trace tree denoted by the pairs of node id and parent id.

```
curNodeId = 0
Init empty T
for d = 0 → MaxDepth do
    for each span s in d^th depth do
        childNum = randomly draw the number of children following {P_d}
        if childNum > 0 then
            for i = 1 → childNum do
                create a new span c_i
                c_i.ID = generate new ID
                update curNodeId
                append [c_i.ID, s.ID] to trace tree T
            end for
        end if
    end for
end for
return T
```

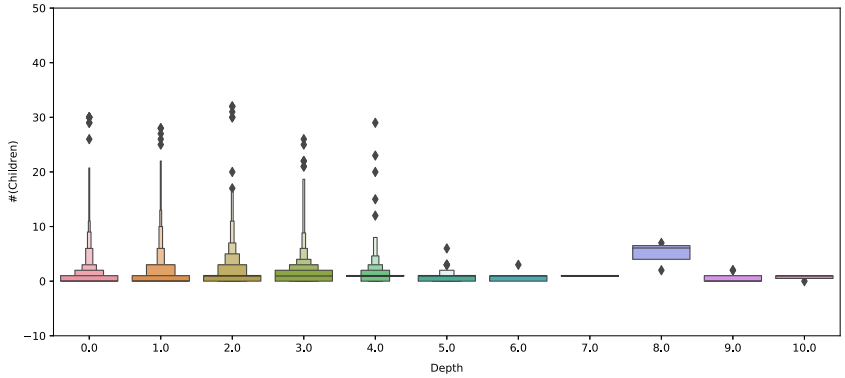
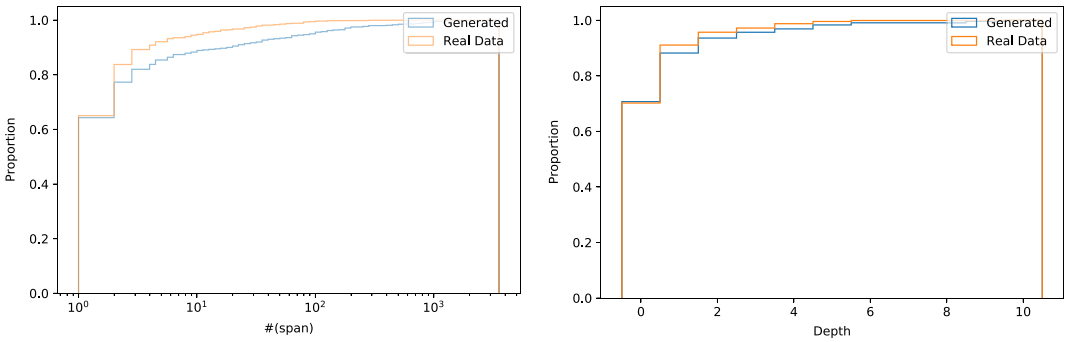


FIGURE 13 The distribution of the number of called children by the component appearing in different depth layer



(A) Distribution of the number of spans in a trace

(B) Distribution of the depth of a trace

FIGURE 14 The comparisons between generated traces with real-world traces

Consistent characteristics. To validate the synthesized result, we compare the size and depth distribution between the generated traces and real-world traces. We repeat the following generation process 1000 times to generate 1000 traces. The distribution comparisons are shown in Figure 14. The result shows that the generated traces have similar statistic characteristics to the real ones.

5.2 | Runtime state generation

Though the trace tree can describe the workflow structure, it still lacks the ability to profile the running time of workflows, as the lack of synchronous/asynchronous information, running time of components, and network communication cost.

5.2.1 | Synchronous and asynchronous calls separation

For separating the synchronous and asynchronous calls for remote calls started by a parent component, the former analysis on the parallel degree, as shown in Figure 7, is an excellent indication to help us to design the separation method. We would wonder if the number of children of a parent node would impact the parallel degree. Thus, we define the parallel rate as the ratio between the parallel degree and the number of total children. We calculate the weighted parallel rate and show the result in Figure 15. There is no obvious pattern between the number of total children and the parallel rate. We conclude that the parallel degree of these calls would not change with the number of remote calls.

Generation method. The highest parallel degree n means that there are at least n asynchronous calls running at the same time. Because the various running time of these n asynchronous calls, these calls finish running one by one, which gradually reduces the parallel degree of the system. This explains why the parallel degree in Figure 7 has continuing value distribution. And the left calls are mainly called as pairs, which makes the majority parallel degree be 2. Applying linear regression to the highest parallel degree to the total number of calls, we set 70% of calls as an asynchronous group. We set left 30% of calls as pairs to run parallelly within pairs and run pair-by-pair serially to simulate the synchronous and asynchronous calls.

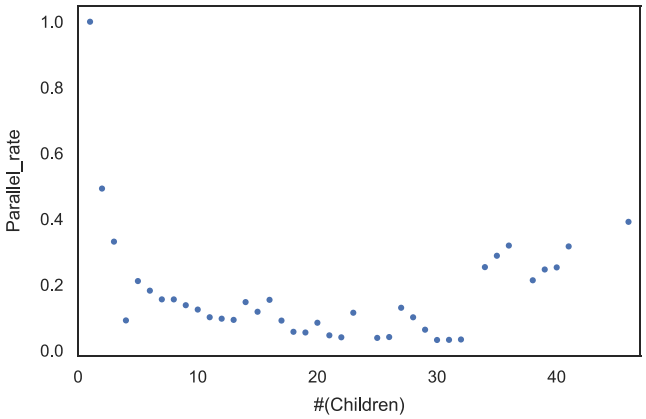


FIGURE 15 The parallel rate under different number of children

5.2.2 | Running time generation

We synthesize the runtime cost from the time dimension. The relationship between running time and resource usage of a component has attached lots of attention before,[22,23] which is out of the scope of our paper. To keep the generation process simple, we use simple equations to help with synthesis. According to the distribution of components' duration, such as the distribution shown in Figure 11A, we set the duration DT of each component in generated workflows to follow the distribution as:

$$\log_{10} DT \sim N(1.85, 0.15)$$

where $N(\cdot)$ denotes the normal distribution. The duration time of each component can be easily simulated by a time burner controlled by the timers in processors.

The network delay is hard to control from the time dimension because it is a dependent variable that varies according to message size. Thus we synthesize this dimension by setting the message size following the distribution as:

$$\log_{10} MS \sim N(3.0, 1.0)$$

5.2.3 | Validations

The former parts describe the generation processes that construct the traces from decomposed characteristics. We would wonder about the effectiveness. Thus, we could further validate the generated results from the overall view, the duration of traces that would be directly impacted by all these characteristics. The duration distributions (1000 traces) are shown in Figure 16, which further proves the representative of these generated traces. The generation process, support data, and the generated traces can be accessed from the open-source repository.[§]

5.3 | Case study

In this section, we further apply the synthesized microservice workflow to performance optimization research. This research intends to explore a better microservice placement schema by co-locating microservices with high RPC costs between them to the same physical machine to improve the running performance of the workflow. We simplify this research problem to find the most effective microservice pairs to co-locate them.

[§]https://github.com/wingwingtwo/TraceCharacterizing.

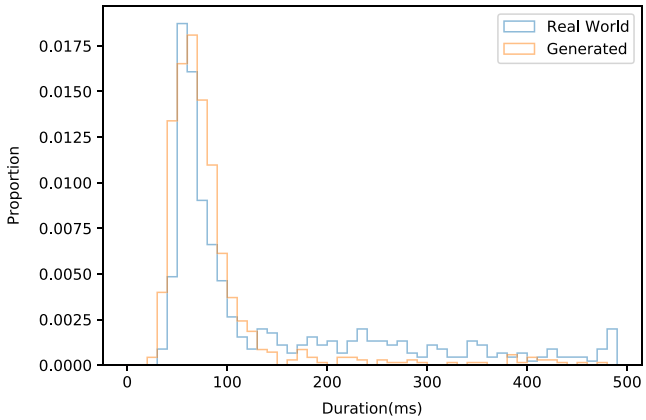


FIGURE 16 The comparison between generated traces and real-world traces on the distribution of trace duration

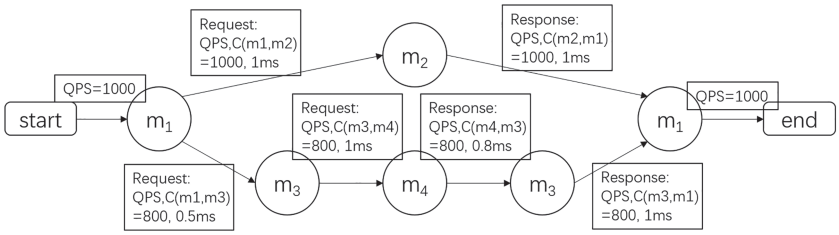


FIGURE 17 A structure example to show why W has better performance than QC design

5.3.1 | Experiment design

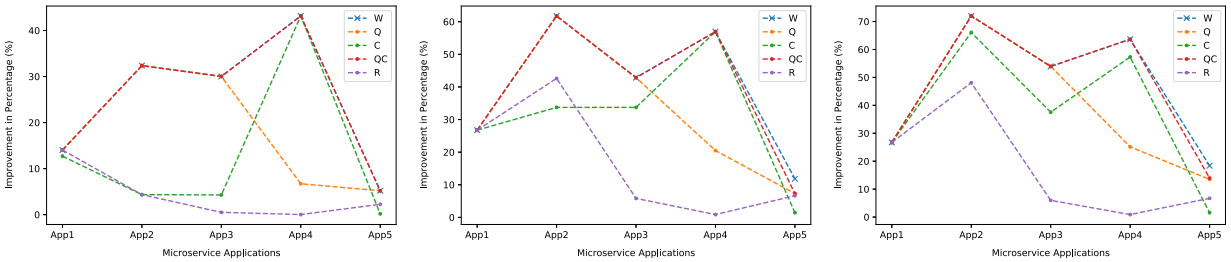
The formerly used weights for finding microservice pairs mainly include the number of invocations and the communication cost.[7,38] Thus we set four designs: (1) C: optimize the most costly pair (the weight is defined by the cost of RPC denoted as *C*)), (2) Q: optimize the most often communication pair (the weight is defined by the throughput denoted as *QPS*)), (3) QC: combine the pair with largest $QPS * C$ value, and (4) R: randomly choose the pairs to optimize as the baseline.

But all these former methods have not considered the impact of workflow structure. For example as shown in Figure 17, critical paths would be an essential impact factor. There are four microservices (m_1,m_2,m_3,m_4): m_1 calls for m_2 and m_3 asynchronously whereas m_3 calls for m_4. According to the QC design, the top pair chosen for optimization is the (m_1,m_2). But the optimization on this pair can not directly affect the workflow's running time, as they are not in the critical path ($m_1 \rightarrow m_3 \rightarrow m_4 \rightarrow m_3 \rightarrow m_1$). We propose the workflow-awareness design (W) to find the pairs in the critical paths with the highest $QPC * C$ value to be the most effective co-location pair. Using W, the top pair chosen for optimization is the (m_3,m_4). Though the improvement is 1.8ms, the optimization affects the workflow directly. For validating the effect of W design, we utilize the synthesized microservice workflow for further exploration.

We deploy our experiment workload to the Cloud instances of ByteDance Cloud. Microservice modules are randomly deployed into two types of Cloud instances: one is 16 cores with 64G memory and another one is 8 cores with 16G memory. The network bandwidth of each host can reach 100Gbps. We run up 5 synthesized applications for this experiment, denoted as "App1" to "App5" with varying numbers of microservices. App1 has 3 microservices, App2 has 5 microservices, App3 has 11 microservices, App4 has 42 microservices, and App5 has 73 microservices. For each application, we warm up the context by 100 repeated requests and record the running time of the following 1000 requests to be the observed data. We optimize the RPC cost of a microservice pair by compiling them together as a single deployment unit, which converts the remote calls to local calls and saves all the RPC cost between them.

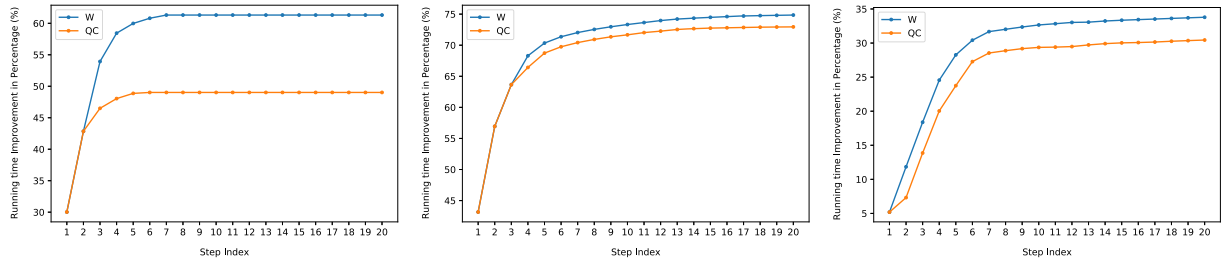
5.3.2 | Experiment result

We denote the optimization effect by the running time improvement in percentage to the original running time for these experiment applications. The improvement of the average running time of these five applications after combining the top one to three most effective pairs is shown in



(A) Optimizing the top one pair (B) Optimizing the top two pairs (C) Optimizing the top three pairs

FIGURE 18 The effect of improvement when optimizing the most effective pairs



(A) App3 (B) App4 (C) App5

FIGURE 19 Comparison between our optimization method W to another good performance design QC on step by step optimization process

Figure 18. We can find workflow-aware design W has the best performance; QC approaches to W; W and QC have better performance than C and Q designs; R has the worst performance. The reason for QC approaching W's performance is that the microservice pair with large throughput and high RPC cost tends to appear in the critical path. The weight design of W contains the considerations of QC design.

For showing the difference between W and QC more clearly, we compare the effect to combine more pairs and introduce a limit not to combine more than four pairs to the same machine to simulate the limited resources of a physical machine. The pair-by-pair optimization effects comparing the QC and W on three applications are shown in Figure 19. We can find that W design has a more significant improvement than QC design. Furthermore, we find the first several steps contribute most of the improvement. In App3, the first 3 steps bring 58.4% improvement whereas the left steps bring 2.9% improvement. In App4, the first 5 steps bring 70% improvement whereas the left 15 steps bring 4.5% improvement. In App5, the first 6 steps bring 30.4% improvement whereas the left 15 steps bring 3.4% improvement. This result recommends that the optimization focusing on top effective microservice pairs would gain a high cost-performance ratio.

6 | DISCUSSION

In this section, we discuss the implications of our analysis to the scheduler design. And we also make a caveat to the generality of the characteristics shown in our study.

6.1 | Implications to microservice deployment

In the production environment, the granularity of the microservices is mainly determined by the functional blocks related to the business logic. To keep the flexibility of microservice architecture and reduce the network cost simultaneously, the performance optimization strategies fall on deployment/placement.

Further exploration of deployment strategies may bring value. Though most of the traces are small in size, the tail latency is determined by the slowest requests that are the traces with RPCs. And the small number of large size traces have a long-duration time and consume a large proportion of CPU resources. These traces cost more than 40% time on network communication. It has great potential to reduce network costs. Except for the optimization from hardware levels,[5,6,26] reducing the number of RPC would take effect on reducing the network overhead and

improve the performance. The strategies on how to better reduce the RPC calls and how many improvements would be brought to performance still need further exploration.

Characteristics of production microservice topology help to benchmark the deployment. Unlike the randomly generated workflows, production workflows have artificial characteristics, such as long-tail distributions of multiple dimensions, a large proportion of small traces, low parallel degrees among remote calls, and wide shapes of traces. Furthermore, lots of performance optimization strategies need to consider the impact of the workflow structures. For example, the wide shape of the trace indicates the possibility of multiple similar critical paths that need to be optimized simultaneously. The synthesized workflows keeping production statistic characteristics can be further used to experiment with exploring the most effective optimization strategies.

6.2 | Limitations

We have to acknowledge that our analysis is limited to a single source of trace collected from ByteDance Cloud. And we are the first one to characteristic the production microservice topology, which makes it hard to cross-validate with other companies.

Tracing function coverage is not 100%. The Toutiao application used for our analysis is the application with the highest coverage, but the coverage rate is still limited to 89%. Other applications of ByteDance (such as Tiktok and Xigua) have not had such high coverage of tracing function, which limits our analysis to these applications.

Business domain is limited. Toutiao (Headlines) is the application to support users to read news, sharing news, advertising, personal recommendation. Though its service target is simple (to read news), the function blocks are complicated, including tens of thousands of microservices. But the fact is the business domain only refers to the news.

The asynchronization/synchronization information is deduced from timestamps. We do not have the direct calling information to deduce the parallel degree of remote calls. Thus we only show the parallel degree for each time slot to reflect the asynchronization/synchronization condition.

7 | CONCLUSION

This paper presents a panoramic view of production microservice workflow structure through an in-depth characterization study on the ByteDance traces. This work is the first literature exposing the production structure of microservice workflow to the research area. The cognition of the microservice workflow provides important guidance on the benchmark design that is vital for further optimization experiments. It limits the exploration boundary of deployment strategies and is the basis for companies to analyze and decide the potential of changing the microservice granularity.

ACKNOWLEDGEMENTS

The authors would like to thank some other people who were critical to this work. They come from the language team of the Production Research and Engineering Infrastructure department of ByteDance. They are Haiping Zhao, Chuansheng Lu, Xiangdong Ji, and Jinzhu Zhang. This work was partially supported by the National Science Foundation of China (No. U20A20173 and No. 62125206).

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in TraceCharacterizing at https://github.com/wingwingtwo/TraceCharacterizing.

ORCID

Yingying Wen https://orcid.org/0000-0001-7469-4066

REFERENCES

1. Services AW. Implementing microservices on AWS. 2019.https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/microservices-on-aws.html
2. Thönes J. Microservices. *IEEE Softw*. 2015;32(1):116.
3. Talwar V. GRPC: A true internet-scale rpc framework is now 1.0 and ready for production deployments. 2016.https://cloud.google.com/blog/products/gcp/grpc-a-true-internet-scale-rpc-framework-is-now-1-and-ready-for-production-deployments
4. Nguyen T. Benchmarking Performance of Data Serialization and RPC Frameworks in Microservices Architecture: gRPC vs. Apache Thrift vs. Apache Avro. *Master's thesis*: Aalto University. School of Science; 2016. http://urn.fi/URN:NBN:fi:aalto-201611025487

5. Lazarev N, Adit N, Xiang S, Zhang Z, Delimitrou C. Dagger: Towards efficient rpcs in cloud microservices with near-memory reconfigurable nics. *IEEE Comput Architect Lett*. 2020;19(2):134-138.
6. Sriraman A, Wenisch TF. µtune: Auto-tuned threading for OLDI microservices. In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). USENIX Association. Carlsbad, CA; 2018:177-194. https://www.usenix.org/conference/osdi18/presentation/sriraman
7. Hu Y, de Laat C, Zhao Z. Optimizing service placement for microservice architecture in clouds. *Appl Sci*. 2019;9(21):4663. https://www.mdpi.com/2076-3417/9/21/4663
8. Wang S, Ding Z, Jiang C. Elastic scheduling for microservice applications in clouds. *IEEE Trans Parallel Distrib Syst*. 2021;32(1):98-115.
9. Yu Y, Yang J, Guo C, Zheng H, He J. Joint optimization of service request routing and instance placement in the microservice system. *J Netw Comput Appl*. 2019;147:102441. https://www.sciencedirect.com/science/article/pii/S1084804519303017
10. Gan Y, Zhang Y, Cheng D, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19. Association for Computing Machinery. New York, NY, USA; 2019:3-18. https://doi.org/10.1145/3297858.3304013
11. Sriraman A, Wenisch TF. Suite: A benchmark suite for microservices. In: 2018 IEEE International Symposium on Workload Characterization (IISWC); 2018:1-12.
12. Zhou X, Peng X, Xie T, et al. Poster: Benchmarking microservice systems for software engineering research. In: 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion); 2018:323-324.
13. Tian H, Zheng Y, Wang W. Characterizing and synthesizing task dependencies of data-parallel jobs in alibaba cloud. In: Proceedings of the ACM Symposium on Cloud Computing, SoCC '19. Association for Computing Machinery. New York, NY, USA; 2019:139-151. https://doi.org/10.1145/3357223.3362710
14. Angius E, Witte R. Opentrace: An open source workbench for automatic software traceability link recovery. In: 2012 19th Working Conference on Reverse Engineering; 2012:507-508.
15. Crawley K. *Getting started with observability lab: Opentracing, prometheus, and jaeger*. Brooklyn, NY: USENIX Association; 2019.
16. Grambow M, Meusel L, Wittern E, Bermbach D. Benchmarking microservice performance: A pattern-based approach. In: Proceedings of the 35th Annual ACM Symposium on Applied Computing, SAC '20. Association for Computing Machinery. New York, NY, USA; 2020:232-241. https://doi.org/10.1145/3341105.3373875
17. Correia J, Ribeiro F, Filipe R, Arauio F, Cardoso J. Response time characterization of microservice-based systems. In: 2018 IEEE 17th International Symposium on Network Computing and Applications (NCA); 2018:1-5.
18. Caculo S, Lahiri K, Kalambur S. Characterizing the scale-up performance of microservices using teastore. In: 2020 IEEE International Symposium on Workload Characterization (IISWC); 2020:48-59.
19. Ueda T, Nakaike T, Ohara M. Workload characterization for microservices. In: 2016 IEEE International Symposium on Workload Characterization (IISWC); 2016:1-10.
20. Gan Y, Zhang Y, Hu K, et al. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19. Association for Computing Machinery. New York, NY, USA; 2019:19-33. https://doi.org/10.1145/3297858.3304004
21. Wu L, Tordsson J, Elmroth E, Kao O. Microrca: Root cause localization of performance issues in microservices. In: Noms 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium; 2020:1-9.
22. Jindal A, Podolskiy V, Gerndt M. Performance modeling for cloud microservice applications. In: Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering, ICPE '19. Association for Computing Machinery. New York, NY, USA; 2019:25-32. https://doi.org/10.1145/3297663.3310309
23. Bao L, Wu C, Bu X, Ren N, Shen M. Performance modeling and workflow scheduling of microservice-based applications in clouds. *IEEE Trans Parallel Distrib Syst*. 2019;30(9):2114-2129.
24. Hindman B, Konwinski A, Zaharia M, et al. Mesos: A platform for fine-grained resource sharing in the data center. In: Proceedings of the 8th Usenix Conference on Networked Systems Design and Implementation, NSDI'11. USENIX Association. USA; 2011:295-308.
25. Callegati F, Cerroni W, Contoli C, Santandrea G. Performance of network virtualization in cloud computing infrastructures: The openstack case. In: 2014 IEEE 3rd International Conference on Cloud Networking (CLOUDNET); 2014:132-137.
26. Brady T, Dongarra J, Guidolin M, Lastovetsky A, Seymour K. Smartgridrpc: The new rpc model for high performance grid computing. *Concurrency Comput Pract Exper*. 2010;22(18):2467-2487. https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1574
27. Jayasinghe M, Chathurangani J, Kuruppu G, Tennage P, Perera S. An analysis of throughput and latency behaviours under microservice decomposition. In: Bielikova M, Mikkonen T, Pautasso C, eds. *Web engineering*. Cham: Springer International Publishing; 2020:53-69.
28. Cheng Y, Chai Z, Anwar A. Characterizing co-located datacenter workloads: An alibaba case study. In: Proceedings of the 9th Asia-Pacific Workshop on Systems, APSys '18. Association for Computing Machinery. New York, NY, USA; 2018. https://doi.org/10.1145/3265723.3265742
29. Lu C, Ye K, Xu G, Xu C-Z, Bai T. Imbalance in the cloud: An analysis on alibaba cluster trace. In: 2017 IEEE International Conference on Big Data (Big Data); 2017:2884-2892.
30. Guo J, Chang Z, Wang S, et al. Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces. In: 2019 IEEE/ACM 27th International Symposium on Quality of Service (IWQOS); 2019:1-10.
31. Ren G, Tune E, Moseley T, Shi Y, Rus S, Hundt R. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*. 2010;30(4):65-79.
32. Kanev S, Darago JP, Hazelwood K, et al. Profiling a warehouse-scale computer. In: Proceedings of the 42nd Annual International Symposium on Computer Architecture, ISCA '15. Association for Computing Machinery. New York, NY, USA; 2015:158-169. https://doi.org/10.1145/2749469.2750392
33. Reiss C, Tumanov A, Ganger GR, Katz RH, Kozuch MA. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In: Proceedings of the Third ACM Symposium on Cloud Computing, SoCC '12. Association for Computing Machinery. New York, NY, USA; 2012. https://doi.org/10.1145/2391229.2391236
34. Wan X, Guan X, Wang T, Bai G, Choi B-Y. Application deployment using microservice and docker containers: Framework and optimization. *J Netw Comput Appl*. 2018;119:97-109. https://www.sciencedirect.com/science/article/pii/S1084804518302273

35. Srirama SN, Adhikari M, Paul S. Application deployment using containers with auto-scaling for microservices in cloud environment. *J Netw Comput Appl*. 2020;160:102629. https://www.sciencedirect.com/science/article/pii/S108480452030103X
36. Rahman M, Iqbal S, Gao J. Load balancer as a service in cloud computing. In: 2014 IEEE 8th International Symposium on Service Oriented System Engineering; 2014:204-211.
37. Kalia A, Kaminsky M, Andersen D. Datacenter rpcs can be general and fast. In: 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). USENIX Association. Boston, MA; 2019:1-16. https://www.usenix.org/conference/nsdi19/presentation/kalia
38. Huang K-C, Shen B-J. Service deployment strategies for efficient execution of composite saas applications on cloud platform. *J Syst Softw*. 2015;107:127-141. https://www.sciencedirect.com/science/article/pii/S0164121215001156

How to cite this article: Wen Y, Cheng G, Deng S, Yin J. Characterizing and synthesizing the workflow structure of microservices in ByteDance Cloud. *J Softw Evol Proc*. 2022;34(8):e2467. doi:10.1002/smr.2467