Delphinus: Improving Resource Efficiency of Applications with Shared Microservices and Diverse Queries

JIUCHEN SHI, Shanghai Jiao Tong University, Shanghai, China and The Hong Kong Polytechnic University, Hong Kong, Hong Kong
JINYUAN CHEN, Shanghai Jiao Tong University, Shanghai, China
QUAN CHEN, Shanghai Jiao Tong University, Shanghai, China
KAIHUA FU, The Hong Kong University of Science and Technology, Hong Kong, Hong Kong
FANRONG DU, Shanghai Jiao Tong University, Shanghai, China
ZIJUN LI, Shanghai Jiao Tong University, Shanghai, China
DEZE ZENG, China University of Geosciences, Wuhan, China
JIANNONG CAO, The Hong Kong Polytechnic University, Hong Kong, Hong Kong
SHUO QUAN, Cloud Computing Research Institute, China Telecom, Beijing, China
JIE WU, Cloud Computing Research Institute, China Telecom, Beijing, China and Temple University, Philadelphia, United States
MINYI GUO, Shanghai Jiao Tong University, Shanghai, China

Microservices are widely shared in production user-facing applications. These shared microservices have various resource usage patterns when queries from different call graphs of different services access them. However, existing microservice management works fail to efficiently scale resources for them, mainly due to the lack of fine-grained scheduling of diverse queries. We therefore propose **Delphinus**, a runtime system that efficiently manages resources for shared microservices while ensuring the Quality-of-Service (QoS). Delphinus comprises a *group-oriented query scheduler* and a *borrowing-based load adapter*. The query scheduler identifies diverse queries, groups the containers of shared microservices, and schedules the queries into separate groups. The load adapter efficiently scales resources for shared microservices, and fully utilizes the idle containers among groups when the loads of diverse queries change. Results show that Delphinus reduces CPU and memory usage by 40.1% and 36.4% for shared microservices, respectively, compared to state-of-the-art works.

This work is partially sponsored by the National Natural Science Foundation of China (62232011), Natural Science Foundation of Shanghai Municipality (25ZR1402241), and National Natural Science Foundation of China (U25B2021).
Authors' Contact Information: Jiuchen Shi, Shanghai Jiao Tong University, Shanghai, Shanghai, China and The Hong Kong Polytechnic University, Hong Kong, Hong Kong; e-mail: shijiuchen@sjtu.edu.cn; Jinyuan Chen, Shanghai Jiao Tong University, Shanghai, Shanghai, China; e-mail: chen_jy@sjtu.edu.cn; Quan Chen (corresponding author), Shanghai Jiao Tong University, Shanghai, Shanghai, China; e-mail: chen-quan@cs.sjtu.edu.cn; Kaihua Fu, The Hong Kong University of Science and Technology, Hong Kong, Hong Kong; e-mail: fukaihua@ust.hk; Fanrong Du, Shanghai Jiao Tong University, Shanghai, Shanghai, China; e-mail: dufanrong@sjtu.edu.cn; Zijun Li, Shanghai Jiao Tong University, Shanghai, Shanghai, China; e-mail: lzjzx1122@sjtu.edu.cn; Deze Zeng, China University of Geosciences, Wuhan, Hubei, China; e-mail: deze@cug.edu.cn; Jiannong Cao, The Hong Kong Polytechnic University, Hong Kong, Hong Kong; e-mail: jiannong.cao@polyu.edu.hk; Shuo Quan, Cloud Computing Research Institute, China Telecom, Beijing, China; e-mail: quansh@chinatelecom.cn; Jie Wu, Cloud Computing Research Institute, China Telecom, Beijing, China and Temple University, Philadelphia, Pennsylvania, United States; e-mail: jiewu@temple.edu; Minyi Guo, Shanghai Jiao Tong University, Shanghai, Shanghai, China; e-mail: guo-my@cs.sjtu.edu.cn.

ACM 1544-3973/2026/06-ART64
https://doi.org/10.1145/3815585

CCS Concepts: • **Computer systems organization → Cloud computing;**

Additional Key Words and Phrases: Cloud, microservices, resource management

ACM Reference Format:
Jiuchen Shi, Jinyuan Chen, Quan Chen, Kaihua Fu, Fanrong Du, Zijun Li, Deze Zeng, Jiannong Cao, Shuo Quan, Jie Wu, and Minyi Guo. 2026. Delphinus: Improving Resource Efficiency of Applications with Shared Microservices and Diverse Queries. *ACM Trans. Arch. Code Optim.* 23, 2, Article 64 (June 2026), 26 pages. https://doi.org/10.1145/3815585

1 Introduction

User-facing applications with stringent **Quality-of-Service** (**QoS**) requirements [3, 35, 49], e.g., Facebook from Meta, have evolved to the microservice architecture. The microservices provide different functions, communicate through the network, and scale independently [9, 33, 57]. While a production application often provides multiple services [22, 30, 46], queries may also go through part of microservices in a service [20, 28, 29]. For instance, Facebook provides *friend recommendation* and *live video* services. In the recommendation service, there are queries of *finding nearby friends* that go through *location* microservice, and queries of *mutual friends* that go through *social graph* microservice. Such an application is often denoted as a graph where nodes and edges are microservices and call relationships [15, 24]. A user query is represented by a subgraph that includes all its involved microservices (namely a call graph [20, 28, 29]).

Figure 1 shows an example application with 2 services, extracted from production microservice traces [28, 31]. Service-2 has 3 call graphs while Service-1 has a single call graph. As observed, microservices *1* and *2* are shared by queries from different services [30, 46], and microservices *3*, *4*, and *5* are also shared by queries from different call graphs in a service [20]. Industrial experience has shown that the load patterns (e.g., queries per second, QPS) of services and call graphs change dynamically [31, 41, 44], shared microservices also have diverse workloads and QoSs when they are called in various services [27, 30]. In this case, a shared microservice uses multiple **Remote Procedure Call** (**RPC**) functions to support various call graphs [31]. Our measurement shows that a shared microservice's CPU and memory usage varies by up to 3.6× and 3.1× across call graphs.

State-of-the-art management solutions for microservices [5, 30, 38, 44, 51, 53], such as FIRM [38], ERMS [30], Nodens [44], have the optimization goal of prioritizing QoS guarantees and then improving resource allocation efficiency, which is the same as our article. However, they neglect the resource usage dynamics for shared microservices, resulting in resource allocation inefficiency. The fundamental inefficiency stems from the heterogeneity of various types of queries, which exhibit varying resource demands and latency QoS targets. Without identifying queries from different call graphs in different services with dynamic loads, they are forced to adopt a conservative strategy to allocate resources based on the maximum demand among all mixed query types. This leads to resource over-provision, resulting in an actual latency far lower than the given QoS target.

Even if diverse types of queries are identified, when diverse queries are served in a shared resource pool, it is inherently difficult for the scheduler to prioritize and allocate resources dynamically to ensure each query type receives its required resources to meet its specific QoS. Without fine-grained resource control, resources like CPU cycles cannot be precisely allocated and isolated per query type in real-time in a mixed serving environment. Our investigation shows that the 99%-ile latency of queries with mixed serving is 3.4× higher than the QoS target, using the lower-bound resource allocation (aggregating minimal resources of queries across call graphs under QoS guarantee). Thus, to ensure that all QoS targets are met, the system has to over-provision resources.

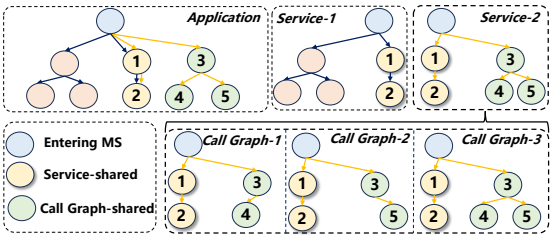


Fig. 1. An example production application with two services. The *Service-2* has three different call graphs.

An intuitive solution is to identify the queries with diverse resource demands, group containers of each shared microservice, and schedule each type of queries to a specific container group. The containers in different groups are co-located on the same hardware, following the standard cloud-native environments [10, 30, 38, 44]. Ideally, grouping diverse types of queries would be done by identifying queries across the call graphs of different services. However, it is not practical since the call graph of a query cannot be known at runtime before this query completes [20, 31, 44].

We observe that, for a user query, the service it belongs to and the RPC function it calls in the shared microservice can be known from its frontend URL and the RPC client of the shared microservices at runtime. We therefore introduce a *two-level runtime query identification scheme* to identify the queries by their service IDs and the called RPC function IDs before scheduling them to shared microservices. This identification allows us to group the shared microservice containers and schedule queries with diverse resource demands to the appropriate container groups.

After identifying diverse queries and scheduling them in groups, it is still nontrivial to achieve efficient resource balancing for shared microservices, because (1) the resources of each container group of a shared microservice need appropriate scaling, and (2) the separate group serving may lead to load imbalance, due to the dynamic loads of services and call graphs. In this case, the queuing on a specific container group brings long tail latency, even though other container groups are idle. A suitable mechanism is required to fully utilize the idle containers across groups of shared microservices during the resource scaling period after the load changes. In this way, the CPU and memory resources can be shared across container groups on the same node.

We therefore propose **Delphinus**, a runtime system for efficient resource scaling of shared microservices while ensuring the QoS. Delphinus comprises a *group-oriented query scheduler* and a *borrowing-based load adapter*. The scheduler utilizes the two-level runtime query identification scheme to identify queries based on their service IDs extracted from the frontend URL and the called RPC function IDs retrieved from the RPC client framework. Based on this scheme, the queries are grouped and scheduled to different container groups at runtime. When loads of diverse queries change, the load adapter scales the resources of each shared microservice group based on monitored loads. Moreover, it adopts a container borrowing mechanism to borrow idle containers across groups and balance the query queue for containers inside a group, to alleviate the long tail latency during resource scaling. This article makes three main contributions.

(1) Investigating the resource inefficiency of shared microservices. We identify the varying resource usage of shared microservices and the resulting resource inefficiency of current methods.

(2) Identifying and group scheduling queries of shared microservices. We identify the queries with diverse resource demands and schedule them to separate container groups, to enable efficient resource scaling.

(3) Efficient resource utilization of shared microservices under dynamic loads. We scale resources based on the monitored loads and propose the container borrowing mechanism to fully utilize idle containers when adapting to the load change of diverse queries.

Table 1. Comparisons between Delphinus and Others on Resource Management for Shared Microservices

	Varying loads of the application	Varying loads of diverse services	Varying loads of diverse call graphs	Diverse query types in shared microservices
Sage, FIRM, Nautilus, and so on [11, 13, 14, 16, 38, 43, 53]	✔			
Wisp, ERMS [30, 46]	✔	✔		
Madu, Derm, Nodens [10, 31, 44]	✔		✔	
Delphinus	✔	✔	✔	✔

We have implemented Delphinus on top of Kubernetes [23], and evaluated it on an eight-node cluster. Results show that Delphinus reduces CPU and memory resource usage by 40.1% and 36.4% on average for shared microservices, respectively, compared to the state-of-the-art works.

2 Related Work

We discuss related works on dynamic microservice architecture and resource management.

Microservice architecture: Most prior works [7, 15, 44, 45, 56] implemented microservice benchmarks with the directed acyclic graph architecture. DeathStarBench [15] consisted of three microservice applications, in which the HotelReservation and SocialNetwork have 4 and 3 services, respectively, with each of 3 shared microservices. Google implemented the Online Boutique [7] with 4 services and 5 shared microservices. There are also some benchmarks [44] with multiple call graphs and microservices shared among them, implemented based on production traces.

Production microservice traces [20, 28, 31] also showed that microservices are shared among different services and call graphs. DGG [12] analyzed the dynamic characteristics within production traces [31] from the dependency graphs, call graphs, and individual microservices, based on which it designs an automated microservice graph generation tool to enable the creation of benchmarks that embody production features. It also reveals the highly shared nature of microservices among different call graphs in production traces, which supports the research context of our article.

Microservice resource management: Some prior works [5, 17, 19, 34, 52, 55] adopted heuristics to scale resources for microservices when the application load changes. Seer [16] and Sage [14] located the bottleneck microservices through machine learning when the application load changes and allocated more resources to them. Some other works [11, 53] also predicted the latency of the microservice application by using deep learning. Moreover, FIRM [38], ELIS [43], and Nautilus [13] analyzed the microservice dependency graph and identified critical microservices for efficient resource allocation at different load levels. The above studies neglect diverse queries from different services and call graphs, which leads to various resource usage of shared microservices.

For diverse loads of different services and call graphs, Wisp [46] limited the rate of some services to maximize the throughput of shared microservices. ERMS [27, 30] adjusted the serving priority of queries from different services in the same shared microservice containers for efficient resource allocation. Moreover, Madu [31] predicted the loads of microservices under varying loads of different call graphs. Derm [10] prioritized allocating more resources to some call graphs to reduce the overall resource usage and QoS violations. Nodens [44] eliminated the execution-blocking effect for fast QoS recovery under call graph dynamics. The above studies fail to identify diverse queries across services and call graphs, and thus cannot allocate fine-grained resources for shared microservices.

Table 1 compares prior work to Delphinus on resource management for shared microservices.

3 Investigating Shared Microservices

In this section, we first characterize shared microservices based on the production-level traces, and then investigate the deficiency of current microservice management schemes.

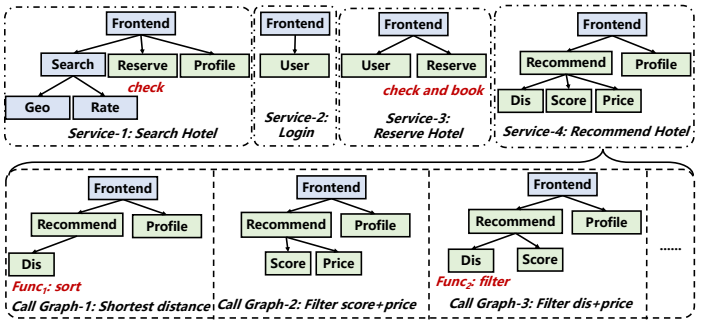


Fig. 2. A simplified HotelReservation application with four services. The Service-4 has multiple call graphs.

3.1 Characterizing Shared Microservices

User-facing applications, such as e-commerce and hotel reservation [7, 15], have queries from users for different functionalities [20, 28, 31]. Figure 2 shows an example HotelReservation application.

To our knowledge, the Alibaba trace [31] is the only public dataset with complete inter-microservice calls, so it is widely used for characterization, benchmarking, and evaluations [30, 40, 44]. This trace is derived from a single application Taobao in the Alibaba Cloud, and thus all observed sharing patterns are inherently intra-application in this subsection. We use it as our primary dataset. The Meta trace [20] also includes partial parent-child microservice calls and is used for supplementary.

3.1.1 Sharing of Microservices. As shown in Figure 2, user queries demand different functionalities, and a query accesses one "service" for the functionality [28, 30, 46]. The queries of a service may go through different parts of microservices in this service, forming different "call graphs" [20, 31, 44]. The queries of *Service-4* generate multiple call graphs owing to various user demands, such as recommending hotels in the closest distance, applying range filtering for distance, score, price, and so on.

Analyzing two production traces [20, 31], most microservices are shared by queries from: (1) different services within the same application and (2) different call graphs within a service, and they are "shared microservices". The "sharing" addressed in our article occurs at the logical and application level. In detail, a single microservice container is shared (invoked) by multiple types of queries (from different services or call graphs), which may impose heterogeneous CPU or memory demand patterns on the same container. In Figure 2, *Reserve* is shared by *Service-1* and *Service-3*, while *Dis* is shared by *Call Graph-1* and *Call Graph-3* of *Service-4*. Some microservices are both shared by services or call graphs in the same service, e.g., *Profile* is shared between *Service-1* and *4*, and among call graphs of *Service-4*.

According to the trace [31], 194 microservices (36.2% of total) are shared by different services, while 101,865,124 queries (42.1% of total) access one or more of them. Also, 74.7% of the microservices are shared by call graphs in the same service with 61.1% of user queries. Our analysis also reveals the existence of dominant microservices. For example, the microservice *MS_37691* is shared by 6 out of the top 10 services. The top services are ranked by their total number of queries in the traces.

Our work aligns with other state-of-the-art microservice resource management works [10, 30, 44], that focus exclusively on resource sharing and management within a single application.

3.1.2 Varying Resource Usage of Shared Microservices. Queries from different services often share microservices but have varying resource usage due to various workload patterns and QoS demands [27, 30]. As shown in Figure 2, *Service-1* calls the shared microservice *Reserve* to check

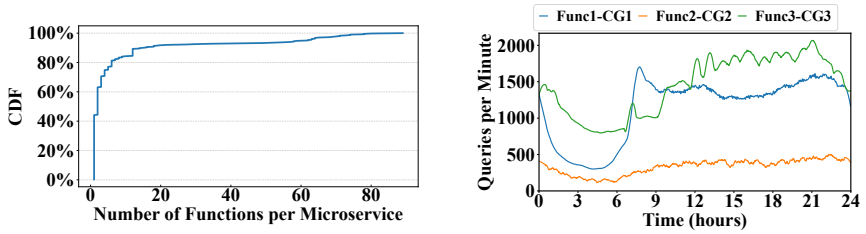


Fig. 3. Trace analysis for the varying resource usage of shared microservices.

room availability while *Service-3* calls it to check availability and proceed with booking. These services have varying workload patterns. Moreover, *Reserve* may have different latency QoSs in different types of queries across services, to meet each service's different end-to-end latency requirements.

Queries from different call graphs within a service may call distinct RPC functions (i.e., API endpoints) of a shared microservice, resulting in varying resource demands. Since our utilized Alibaba trace [31] provides the function interface names and response times while the actual workloads are unavailable, we conduct analysis to collect the RPC function number for shared microservices of the trace. As shown in the left part of Figure 3, we observe that nearly 60% of the shared microservices handle 2 or more distinct RPC functions to handle different call graphs. Different function names suggest they implement different functionalities, which in turn are highly likely to exhibit different resource usage patterns (e.g., CPU-bound vs. I/O-bound). Similarly, Meta trace analysis also stated that different queries may trigger different endpoint functions [20].

The right part of Figure 3 also shows that the loads of 3 example RPC functions invoked by different call graphs change dynamically over time. Similarly, production studies showed that loads of different services and call graphs have high load fluctuations over time [30, 31, 44]. Our investigations of the top 20 services by total query count in Alibaba traces show that the average load variations of call graphs are 38.6% every 5 seconds. With the varying loads of call graphs that invoke different RPC functions, the resources of shared microservices show dynamic demands.

3.2 Low Resource Allocation Efficiency

We investigate the resource allocation efficiency of existing microservice management schemes.

Three benchmarks in prior studies [15, 44] (**HotelReservation** (***HR***), **SocialNetwork** (***SN***), and *EB1*), and an industrial-level benchmark Google **Online Boutique** (***OB***) [7] are used. *EB1* is derived from the Alibaba microservice by prior work [44]. It exhibits characteristics of relatively complex and production-level microservices, comprising 1 service, 5 call graphs, 21 microservices, and including 5 microservices shared among queries with different call graphs. *OB* has 4 services and 5 shared microservices. The HR and SN include service-shared microservices. In detail, HR comprises 4 services, among which 3 microservices (*reserve*, *user*, and *profile*) are shared across queries from all 4 services. SN consists of 3 services, where 3 microservices (*post-storage-service*, *user-timeline-service*, and *home-timeline-service*) are shared across queries from all 3 services. The above benchmarks also reveal some existence of dominant microservices similar to our data analysis in Section 3.1.1. For the example in EB1, the "ms-14" is shared by 4 out of the total 5 call graphs within that benchmark.

Following previous work [44], to simulate real-world applications, we enhance *HR* and *SN* by adding call graph sharing features. In the HR, we extended the "hotel recommend service" by adding 4 microservices that are shared across 7 call graphs, enabling support for hotel filtering based on distance, price, and score. In the SN, we extended the "compose-post service" by introducing 2

Table 2. Experiment Specifications

	Specifications
Hardware	Eight-node cluster, each node is a dual-socket DELL PowerEdge R730 server Each node has two Intel Xeon E5-2630 V4 processors Each node has 10 physical cores per CPU (20 in total), CPU governor with on-demand, Each node has 4 channels per CPU (8 channels total) with 256GB of DDR4 RAM, 1Gbps Local Area Network (LAN) bandwidth among nodes
Software	Ubuntu 20.04.2 LTS with kernel 5.11.0-34-generic, Docker version 20.10.18, Kubernetes version v1.20.4, Standard Linux kernel network stack (no kernel-bypass), Default NUMA policy (no explicit optimizations or bindings)

shared microservices that are utilized by 6 call graphs, to support various methods of text filtering and image compression. These extensions were implemented within each benchmark's original framework (e.g., gRPC for microservice communication). The performance difference between query types is substantial in our testing benchmarks. For shared microservices across different call graphs, we observe an average variation in CPU demand of 3.8 times and in memory demand of 3.1 times. Their execution latencies also differ by an average of 3.6 times.

We utilize ERMS [30], a resource management system for shared microservices, as the representative baseline. ERMS can identify queries from different services, but does not identify different call graphs of services and queries from them, instead merging the call graphs to manage their resource allocation as a whole. Moreover, ERMS conducts mixed serving in containers of shared microservices for queries from different services and call graphs. Table 2 summarizes the hardware and software configurations. In all test cases, the initial load is 70% of the supported peak load under ERMS [30], and the shared microservices are allocated the minimal required resources when the query proportions of different call graphs in different services are the same. To form various scenarios of load proportions of diverse queries, we randomly increase or decrease the loads of different numbers of services and call graphs by 30%. We test the *HR* and *SN* benchmarks under 16 different scenarios, and the results of other scenarios and benchmarks are shown in Section 8.

We define the *Lower-Bound* resource allocation, which is achieved by separately profiling the minimal CPU and memory demand of each self-contained call graph in isolation to ensure the QoS in a brute-force way for each test case. This definition isolates the intrinsic resource demand of each type of queries with each call graph, providing a clear and reproducible standard to evaluate resource allocation efficiency, which quantifies the potential gain from perfect query-type-aware isolation. Moreover, the call graphs in our benchmarks comprise self-contained microservices where each container encapsulates the entire computations for its function, leaving no unmonitored backend threads. This aligns with related microservice resource allocation studies [10, 30, 44], ensuring that the CPU and memory usage we profile for a call graph represents its total resource demand.

Figure 4 shows the CPU and memory usage of ERMS normalized to *Lower-Bound*. In all test cases, both ERMS and Lower-Bound can ensure the QoS (99%-ile latency) of diverse types of queries. The x-axis represents the number of services and call graphs whose loads are changed compared to the initial state. For instance, *(3,7)* means the loads of queries from 3 services and 7 call graphs are changed compared to the initial state. As observed, ERMS uses 56.1% and 53.7% more CPU cores and memory space, proving the low resource efficiency.

3.3 Diving into the Underlying Reasons

The low resource efficiency is caused by two reasons: *ignoring diverse types of queries* and *resource contention in mixed serving*.

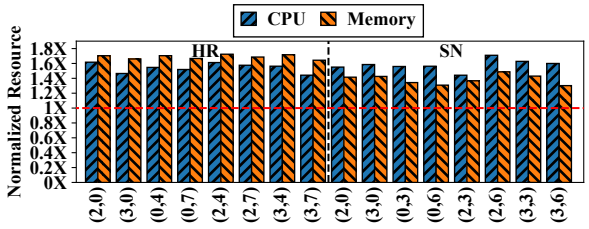
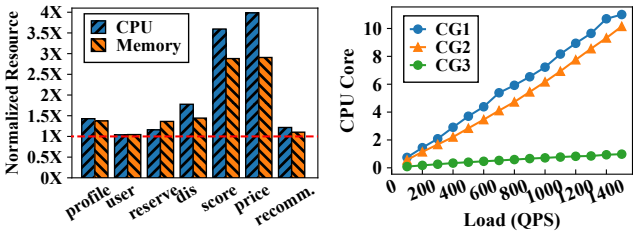


Fig. 4. The CPU usage and memory usage of ERMS normalized to the *Lower-Bound* resource allocation.



(a) Normalized resource allocation. (b) An example CPU core demand.

Fig. 5. The resource allocation of ERMS for each shared microservice. The *recomm.* is short for *recommend*.

3.3.1 Ignoring Diverse Types of Queries. Current works cannot identify diverse queries from each call graph of each service, resulting in resource allocation inefficiency. We take the scenario *(3,7)* of the *HR* as an example here. Other test cases show similar observations.

Figure 5(a) shows the resource allocation of ERMS normalized to the Lower-Bound allocation for each shared microservice. We can first observe that ERMS allocates more CPU resources to each shared microservice, especially for the *dis*, *score*, and *price*, which are shared by call graphs in a service. This is because ERMS cannot identify among queries from different call graphs, and due to the dynamic nature of call graph loads over time [31, 44], their resource allocation needs to follow the highest values from historical data. For instance, Figure 5(b) shows the CPU demand of *price* across call graphs. The steepest slope curve is selected in ERMS for allocation.

Figure 5(a) also shows that ERMS uses more memory space for all shared microservices, especially for *dis*, *score*, and *price*. This is because each type of queries in these microservices has the maximum supported load of a single container. When not identifying queries across call graphs, horizontal scaling has to be conducted based on the type of queries with the lowest supported load to ensure QoS for all queries, leading to more startup containers with more memory usage. The resource allocation difference between ERMS and Lower-Bound of the *recomm.* (which is also shared by call graphs) is smaller, as this microservice's resource usage variations are minimal across call graphs.

The merging of call graphs of ERMS [27] also leads to higher resource allocation. For instance, *price* only appears in some call graphs of *Recommend hotel* service, but it has to be allocated resources with the service's total load. Moreover, although still worse than Lower-Bound, we can find that ERMS has relatively good efficiency for *profile*, *user*, and *reserve* that are shared among services. This is because ERMS can identify queries across services.

3.3.2 Resource Contention in Mixed Serving. Even if diverse types of queries can be identified, the mixed serving of them requires complex scheduling policies to decide the serving priority

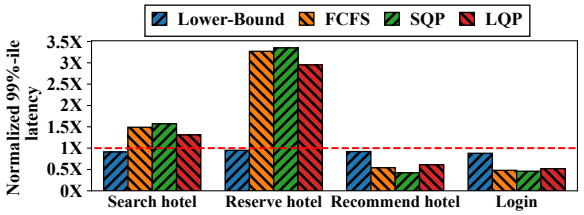


Fig. 6. The 99%-ile latency of Lower-Bound, FCFS, SQP, and LQP of four services normalized to their QoS.

of each type of queries. Except for the **First Come First Served** (**FCFS**) scheme of the query mixed serving in shared microservices, current work [30] uses Linux tc [48] to prioritize serving queries from the service with the shortest latency QoS. In this subsection, we evaluate the resource efficiency of three mixing schemes, including FCFS, **shortest QoS priority** (**SQP**), and **longest QoS priority** (**LQP**). We also take the *(3,7)* of the *HR* benchmark as an example.

We first allocate the container number and CPU cores to each shared microservice for the three mixing methods according to Lower-Bound's resource allocations. Figure 6 shows the 99%-ile latencies of *HR*'s four services of Lower-Bound and the three mixing schemes. We can observe that all mixing schemes have QoS violations in two services. Since the QoS and resource demands of various types of queries are different, each type of queries is hard to obtain the required CPU cores that can ensure its QoS. When resource allocation of shared containers is tight, some types of queries may not contend for enough resources and thus result in QoS violations. We further try to increase the resource allocation for microservices in a brute-force way to force QoS compliance for all services. Evaluation results show that, compared to the Lower-Bound, FCFS, SQP, and LQP require 22.4%, 22.9%, and 21.6% more CPU cores, while the values for the memory are 23.2%, 23.6%, and 22.5%.

The above analysis illustrates that we can improve resource efficiency by scheduling diverse types of queries to different container groups and managing resources for each separately.

3.4 Design Requirements of Delphinus

We design **Delphinus** to resolve the problems listed above. It schedules different types of queries to separate container groups and scales their resources independently. However, it is nontrivial to effectively group and schedule diverse queries.

Firstly, it is infeasible to identify diverse queries and schedule them into different container groups for each call graph in each service. This is because the call graph of a query that arrives at the shared microservice is not known at runtime before it is completed [20, 31, 44]. **Delphinus requires a feasible scheme to identify the queries with diverse resource demands and schedule them with retrievable information at runtime.**

Secondly, when the loads of diverse queries change, the resources of shared microservices should be scaled accordingly. Moreover, the isolated grouping may lead to imbalanced resource usage among groups, leading to query queues and long tail latency. Figure 7 shows an example shared microservice that serves two types of queries A and B in isolated groups. When loads of A and B increase or decrease, they need horizontal scaling (generate or recycle some containers), respectively. In this case, Group A's containers are overloaded, while Group B does not fully utilize its idle containers. **Delphinus needs to obtain loads of diverse queries for efficient scaling and fully utilize idle containers among groups to serve overload queries during the scaling.**

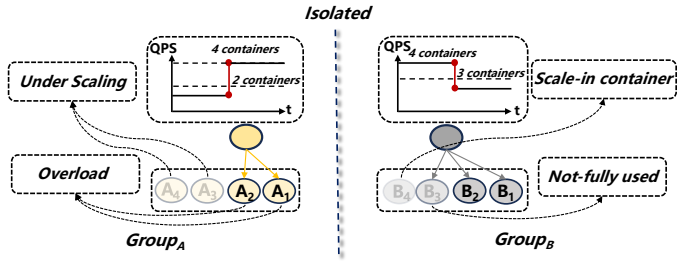


Fig. 7. Serving two types of queries in isolated groups.

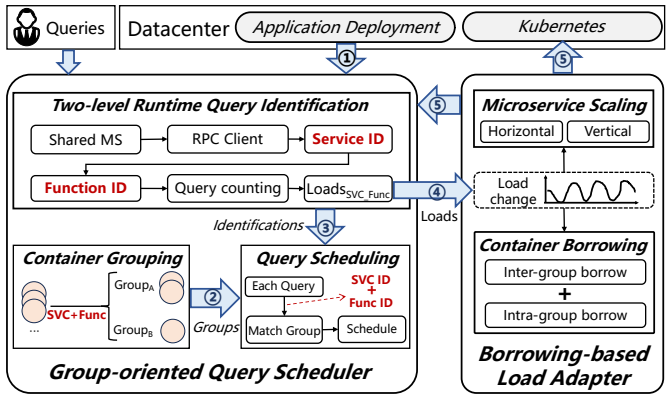


Fig. 8. System overview of delphinus.

4 Delphinus Methodology

Figure 8 shows the design overview of Delphinus. It comprises a *group-oriented query scheduler* and a *borrowing-based load adapter*. The query scheduler identifies the queries under two levels of the service that a query belongs to, and the RPC function it calls at runtime, and monitors their fine-grained loads. Based on query identifications, the containers of shared microservices are grouped and queries are scheduled to different container groups. The load adapter periodically rescales the resources of shared microservices when the loads of diverse queries change. Moreover, a container borrowing mechanism is enabled to fully utilize idle containers to handle overload queries during horizontal scaling.

Although the call graph of a query is unknown before it completes, queries across call graphs could call different RPC functions of the shared microservices that can be known at runtime. Thus, we propose a two-level runtime query identification scheme to identify queries with diverse resource demands by their service IDs and called RPC function IDs. Based on this scheme, the queries are scheduled to separate container groups. The real-time load of each RPC function of each service is also monitored by using the query counting from the RPC client framework (Section 5).

When loads of diverse queries change, according to the monitored loads, the adapter scales horizontally (generates or recycles containers) for shared microservice groups on top of Kubernetes [23] with the container number minimized to reduce memory usage, and scales vertically to allocate the minimal required CPU cores for each shared microservice container. Also, since the imbalanced resource usage during the scaling, we propose a container borrowing mechanism to

borrow inter-/intra-grouped idle containers to serve overload queries, to reduce queued queries and the tail latency (Section 6).

Specifically, Delphinus works in the following steps for an application. (1) It obtains the services, call graphs, and shared microservices with resource demands from historical online deployment data. The history of call graphs and resources is collected using Jaeger [21] and Prometheus [47], respectively. (2) The query scheduler groups shared microservice containers according to the service and RPC function IDs. (3) When queries arrive at the RPC client of the shared microservice, the query scheduler identifies them based on their service and function IDs and schedules them to the appropriate container groups. (4) Meanwhile, to get the loads of diverse queries, a daemon process that periodically retrieves the query counting data from the RPC framework is initialized on the master server. (5) When loads of diverse queries change, the master server runs the adapter to scale shared microservices' resources, activates the container borrowing to fully utilize idle containers, and updates the container numbers for shared microservice groups into the query scheduler.

5 Group-Oriented Query Scheduler

In this section, we introduce the two-level runtime query identification scheme, as well as the container grouping and query scheduling based on this scheme.

5.1 Two-level Runtime Query Identification

We identify the queries under the levels of the service a query belongs to and the called RPC functions. Meanwhile, the fine-grained loads of them are monitored by querying counting from the RPC client framework.

5.1.1 Identifying Queries from Diverse Services. Since queries from different services may have different computing resource demands for the shared microservices due to different workload patterns and QoS requirements, we first identify queries from diverse services.

The number of services is generally limited in production applications [30]. Also, the HTTP URLs which are used to invoke the entering microservice from different services are conveniently identified [15], e.g., an example HTTP URL for *SVC*1 service is "http://entering-ms-ip:port/SVC1?params". Thus, our design tries to automate identifications for queries from various services at the entering microservice and pass identification results through subsequent RPC calls.

As shown in Figure 9(a), for queries from different services entering into the entering microservice, we first automatically extract the service IDs of them from the distinct HTTP URLs. Then, we embed the corresponding service IDs in the RPC query headers (e.g., metadata in the gRPC context [18]) for the transmission across subsequent RPC calls among microservices. All the above operations are encapsulated into an external link library and can be activated by adding one line of code in the entering microservice. When the query arrives at the Balancer of the RPC client of the shared microservice, we retrieve the service ID from this query's RPC header, which will be used to decide which shared microservice's container this query will be sent to.

Moreover, we deploy a query counter for different services inside the Balancer of the RPC client of each shared microservice, e.g., microservice-4's counter is deployed in the microservice-3's RPC client. We monitor the status of different queries from the Interceptor of the RPC client, and when a query's status changes to ready, we increment the counter of its corresponding service. The query identification counters of different services are first obtained ($Counter_{SVC}$).

5.1.2 Identifying Queries Invoking Diverse Functions. Since the call graph of a query cannot be known at runtime until it completes [20, 31, 44], it is infeasible to identify the queries at runtime directly at each call graph granularity for the microservices shared by call graphs in a service.

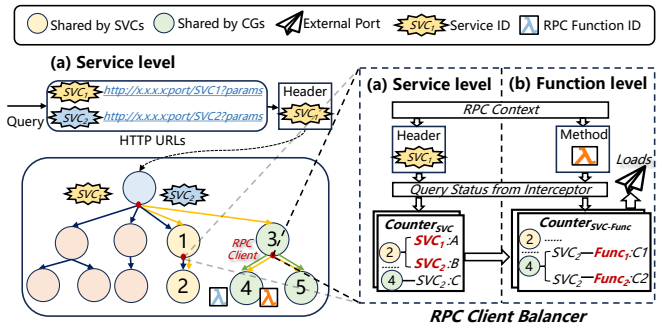


Fig. 9. The two-level runtime query identification scheme.

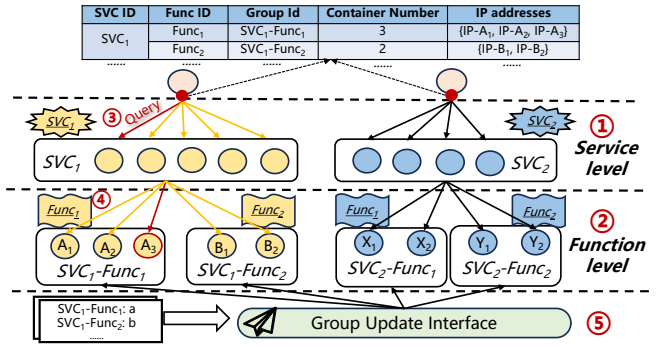


Fig. 10. Container grouping and query scheduling.

As the queries across call graphs could invoke different RPC functions of shared microservices with varying resource usage, we identify the queries for each RPC function in each service, as shown in Figure 9(b). In detail, for a query that arrives at the RPC client of the shared microservice, we retrieve its invoked RPC function ID from the context information in the RPC client framework (e.g., the *FullMethodName* in gRPC [18]). Moreover, we adopt the same method to Section 5.1.1 to count the query number of different RPC functions in each service ($Counter_{SVC-Func}$).

An external network port is offered for obtaining the counting data of each RPC function in each service. Delphinus also periodically retrieves the query counting data, to monitor the load of queries for each RPC function in each service. The fine-grained loads will be utilized for efficient resource utilization of shared microservices when loads of diverse queries change (Section 6).

5.2 Container Grouping and Query Scheduling

Enabled by the two-level query identifications, we aim at serving queries of shared microservices at the service and RPC function levels, to avoid the mixed serving of different types of queries.

As shown in Figure 10, we first group the containers based on the service IDs for the shared microservices (①). For each service group, we then group the containers of each service with different RPC function IDs (②). Therefore, the shared microservice's containers are grouped by the combination of the service ID and RPC function ID. For each container group, Delphinus maintains the container number and corresponding IP addresses for the containers.

When a query arrives at the RPC client of a shared microservice, Delphinus initially matches the ID in the query's RPC header (added by service identification in Section 5.1.1) with the group

ID SVC_i (③), and then gets the called RPC function ID $Func_i$ from the context information in the RPC client framework (④). A container in the group SVC_i-$Func_i$ will be chosen to serve this query. Within each group, the containers are sorted in ascending order by their IPs, and we schedule queries according to this order to fill up each container, i.e., start to schedule to the next only when the previous one has queued queries in the RPC client. The purpose of this is to minimize the mixing of diverse queries in a single container during the container borrowing process (Section 6.2). The above operations are integrated into the RPC client's Balancer without modifying the application.

After the horizontal scaling from the load adapter (Section 6.1), the containers of each group need to be updated. The query scheduler offers a group update interface to receive the new container numbers for each group (⑤). After obtaining the updated group container numbers, the query scheduler first re-acquires the IPs of all connected containers of the shared microservice, and then dynamically adds or removes IP addresses of containers in different groups.

It is worth noting that, regardless of whether a microservice is shared among services, call graphs, or both, Delphinus grouped schedules its queries under the same process shown in Figure 10. When a microservice exhibits both types of sharing, mixed serving leads to worse QoS and resource efficiency. Moreover, memory utilization does not benefit from sharing in such cases, since horizontal scaling should be based on the lowest supported load per query type, requiring lots of containers.

6 Borrowing-based Load Adapter

In this section, we introduce two adaptations when the loads of diverse types of queries change. The first is resource scaling for shared microservices, to improve the resource allocation efficiency of shared microservices. The second is the container borrowing to fully utilize idle containers across isolated groups, which aims at reducing the long tail latency during horizontal scaling.

6.1 Resource-efficient Microservice Scaling

Horizontal and vertical scaling respectively allocate the minimal required containers and CPU cores for each group of shared microservices based on the monitored loads from query identifications (Section 5.1), while ensuring the QoS. The QoS breakdown for each microservice is set based on prior works [30, 54] that enabled the least total resource allocation through an optimal QoS division.

Horizontal scaling is required as a container has a maximum-supported load, specified by the maximum concurrency of its RPC server. For example, in the gRPC [18], this parameter is configured by the developer using the *grpc.MaxConcurrentStreams()* option, which is passed as an argument when initializing the gRPC server. Developers often set a conservative value not primarily to define the absolute hardware capacity, but to enforce fault isolation [1, 36, 39]. For stateful backend microservices (e.g., databases, caches), the concurrency safety for accessing these shared resources is inherently provided by the backend microservices through their client APIs (e.g., atomic operations like *InsertOne()*) and internal concurrency control [4, 32], which do not introduce additional concurrency constraints that need to be managed. Delphinus first determines the minimum container number and then scales each container's CPU cores vertically.

Horizontal Scaling: Its decision is conducted for each container group of each shared microservice, in which a group serves queries that belong to a specific service and call a specific RPC function. For each shared microservice, we calculate the container number of its $Group_i$ by its monitored load $monitor\_load_i$ divided by the maximum load supported by one container max_load_i as

$$Number_i = \lceil \frac{monitor\_load_i}{max\_load_i} \rceil. \tag{1}$$

The $monitor\_load_i$ is periodically obtained by the query identifications (Section 5.1). The max_load_i (i.e., QPS) for a container is influenced by the underlying hardware and software stack. We learn

this value from historical deployment data on the target cluster, under the practical assumption that the environment is relatively consistent, like other studies [10, 30, 38, 44, 53]. Delphinus can also adapt to environmental change, as the max_load_i is not a static configuration. If the underlying hardware or software changes, our continuous online learning mechanism will incorporate new performance data, allowing us to gradually relearn the new effective capacity of the containers.

Horizontal scaling is then conducted in parallel for shared microservices to reduce the scaling overhead. To further reduce the latency of cold starts for enabling horizontal re-balancing, Delphinus includes the container borrowing mechanism that will be introduced in Section 6.2. Following prior works [10, 30], since microservices are basically sensitive to CPU resources and have relatively steady memory usage, we allocate each microservice container with fixed memory space.

Vertical Scaling: As queries served in different container groups of a shared microservice can have various CPU core demands, we establish a prediction model between the load size (QPS) and resource demand for each container group of each shared microservice to ensure the QoS. Prior works have demonstrated that the resource prediction of microservices has a linear relationship with load size with popular microservice benchmarks on their specific hardware [27, 31, 44]. We therefore predict the CPU core demand from the load size of each container group in each shared microservice by using an individual **Linear Regression** (**LR**) model.

We calculate the total CPU cores for each $Group_i$ of each shared microservice as

$$Tot\_Res_i = a_i \times monitor\_load_i + b_i. \tag{2}$$

In this equation, a_i and b_i represent the slope and intercept of the LR model of the container group. Corresponding to filling up each container of query scheduling (Section 5.2), the last container of the group may not be fully utilized. Therefore, for m containers in a group, each of the first $m-1$ containers is allocated with the maximum CPU cores that it can use, and the last one is allocated with the remaining resources. Similarly, we also vertically scale the minimal required memory for the last container in the group according to its realistic demand. In this way, the allocated resources are fully utilized to serve queries. Moreover, when the total allocated CPU resources are tight and a group requires vertical scale-up, Delphinus first identifies under-utilized groups that can safely scale down, reclaims their idle CPU quotas, and reallocates them to the high-demand group, to enable the vertical balancing among groups.

To obtain the LR models, for each call graph of a benchmark, we profile its resource demand at 20 evenly spaced loads (from 0 to the system's maximum capacity). Each load is run for 10 seconds, thus the per call graph profiling time is 200 seconds. The training of the LR model is highly efficient with less than 150ms per model. Thus, the total profiling time and training data volume are respectively 33.3/26.7/16.7/13.3 minutes and 48.2/38.5/24.1/19.3KB for HR/SN/EB1/OB. During online testing, these models achieve high prediction accuracy: 97.1%/95.9%/93.8/98.7% for HR/SN/EB1/OB on average. The datasets we used for training and for testing are separate and non-overlapping. We define a test load point as *similar* to any point in the training dataset if their relative error is within 10%, and the percentage of similar points is as low as 5.4% on average.

For adapting to online load changes, we employ a continuous online retraining strategy with a sliding data window. Each group's LR model is retrained every minute using the most recent historical data (e.g., past 5 minutes). This update allows the model to adapt to changing loads. Moreover, we handle the potential prediction errors as follows: First, for prediction higher than actual demand, since this does not violate QoS, we do not actively correct it, which has a similar solution to previous work [31]. Otherwise, we actively monitor each microservice's latency using Jaeger [21]. If the latency exceeds the QoS, we trigger a feedback-driven correction, i.e., the resource for that microservice is incrementally increased (e.g., by 5% steps) until the QoS is satisfied. This mechanism is a standard design, consistent with systems like prior studies [5, 38, 42].

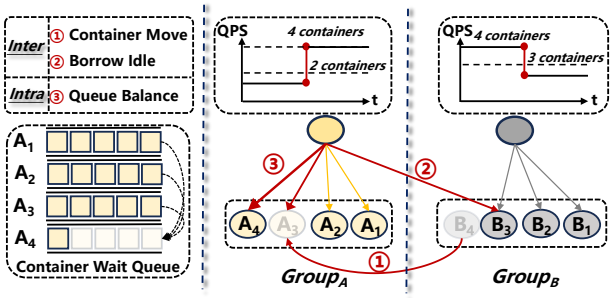


Fig. 11. The inter-/intra-group container borrowing.

6.2 Container Borrowing Mechanism

When the loads of queries from some services or call graphs increase, the resource allocation for some groups of shared microservices is insufficient and queries begin to queue. Due to the isolation among groups, some other groups that have idle resources cannot be used, leading to imbalanced container usage. Since the time overhead of resource scaling (especially scaling-out containers) is non-negligible [26, 50], this can lead to lots of queued queries and long tail latency during the scaling. As shown in Figure 11, we therefore design the container borrowing mechanism.

For inter-group container borrowing, we directly move the idle containers from the groups whose loads decrease to the groups whose loads increase (①). This can effectively reduce the overhead to start new containers, thereby reducing query queuing. The container moving among groups is realized by changing the routing of different queries to different sets of containers. Secondly, we borrow containers with idle resources from other groups to serve queries from the groups that require scaling out (②), to reduce the number of queued queries. As we fill each container in a group in order during query scheduling, one container in each group may have spare capacity. A borrowed container is utilized until it is fully depleted, i.e., the query queue starts to occur for it.

For intra-group container borrowing, we schedule newly-arrived queries to the containers in a group in a query queue-balanced manner when the loads of this group are excessive (③). Specifically, we monitor the query queue lengths of different containers inside a group at runtime and prioritize scheduling the next query to the container with the shortest query queue. The queue length is used only within groups serving a single query type, where execution latency is highly consistent. The Coefficient of Variation of latency within groups is only 1.71%. Thus, within this homogeneous context, queue length is a direct and reliable measure of remaining workload. In this way, containers with long queues can quickly drain queued queries, and all containers inside the group are fully utilized. This decreases the queued query draining time and tail latency after the loads change.

Taking examples in Figure 11, $Group_A$ needs to scale-out to 4 containers while $Group_B$ scales-in to 3 after loads change. The container B_4 is directly move from $Group_B$ to $Group_A$ and $Group_B$'s last container B_3 can also be borrowed by $Group_A$. After A_4 is started, more queries are scheduled to it as the old containers already have lots of queues, until the queued queries are balanced.

Each part of this mechanism is low-overhead and activated only in its specific scenario. In detail, ① activates only when a group has completely idle containers (i.e., after load decrease), ② activates only when a group has containers operating below capacity, and it provides temporary capacity to an overloaded neighbor group; ③ activates only after a group scales up with new containers, to quickly distribute loads of queries. The overhead for each part is negligible, as it involves only updating the routing of queries to different containers, which takes less than 0.15ms for each query.

Moreover, we measured the overhead of routing changes for the potential loss of data locality (e.g., cache state not being available in the new container). Results show that it introduces only 4.8ms of additional latency on average across our benchmarks. This is negligible compared to typical query execution time (hundreds of milliseconds). The overhead is low since: (1) containers are typically deployed within the same cluster with sufficient network bandwidth in the cloud [10, 30, 44]. (2) most microservices are stateless [28, 31] in production cloud, thus data locality is not a concern.

7 Implementation of Delphinus

For query scheduling, Delphinus extracts the service ID from each query's HTTP URL and embeds it into the RPC query header. It is encapsulated into an external link library and activated with just one line of code in the entering microservice. Delphinus utilizes persistent gRPC connections, which are consistent with standard microservice communication practices. These persistent connections are managed within the gRPC client-side load balancing framework. We have implemented a custom Picker interface [6, 37] in the Balancer of the RPC client framework [18]. It is not implemented as a separate microservice, but to replace the default scheduling policy (e.g., round-robin) to realize group scheduling of diverse queries. This implementation has little application code invasiveness.

Regarding load balancing, our custom Picker first routes different types of queries to their respective container groups, and then performs load balancing within each group by selecting an appropriate server from the available connection pool. Since HTTP/2 used by gRPC supports multiplexing, multiple queries can be processed concurrently over a single connection. This ensures that the connection persistence does not constrain load balancing decisions, i.e., the Picker can freely direct new queries based on real-time load, regardless of existing long-lived connections.

For the load monitoring, we collect load metrics by querying the counter API within the RPC client framework (Section 5.1) every 5 seconds. This operation is lightweight with an overhead of about 7ms per sampling interval. The choice of a 5-second interval is justified since: (1) existing systems typically perform reactive resource adjustments at the scale of tens of seconds to minutes [10, 30, 31]. (2) Our analysis on Alibaba traces [31] shows that the load change at a 5-second interval is 38.6%, whereas it drops to 28.1% at a finer 1-second interval. This indicates that 5 seconds effectively captures the relevant dynamics without being overly sensitive to noise. Moreover, the monitoring is designed for our scheduling granularity, i.e., the aggregate load for each container group.

For the microservice scaling, Delphinus re-allocates resources based on periodically monitored microservice loads like prior works [30, 38, 44]. It is built on **Kubernetes** (**K8S**) [23] with horizontal scaling (generating or recycling containers) of microservices enabled by the K8S deployment module. For vertical scaling (adjusting CPU and memory for containers), Delphinus uses Linux cgroups [8] like prior works [25, 43, 53]. Moreover, we adopt the resource-balanced mapping method from ERMS [30], which tries to equalize resource utilization across all servers. This approach effectively prevents any single server from becoming overloaded quickly. The server-microservice mapping does not affect Delphinus's effectiveness in optimizing resource efficiency for shared microservices. This is because the mapping operates at the deployment level while Delphinus's optimization targets runtime query scheduling. These two aspects of optimizations are orthogonal.

8 Evaluation of Delphinus

In this section, we first evaluate Delphinus's resource efficiency under the QoS guarantee, then explore each module's effectiveness and its stability on production traces.

8.1 Evaluation Setup

Table 2 shows the experimental platform. We evaluate the *HR*, *SN*, *EB1*, and *OB* benchmarks on an eight-node cluster. Five nodes deploy microservices, one node hosts the K8S master and Delphinus,

and two nodes generate loads. Following prior microservice management works [30, 44, 53], the QoS targets of different services in our benchmarks are set in the range of 50ms to 300ms.

Same to Section 3.2, the initial total load is 70% of the supported peak load of our cluster, and shared microservices are allocated the minimal required resources when the query proportions of different call graphs in different services are the same. The "minimal required" refers to the minimum CPU and memory allocation required for a microservice to meet its QoS when profiled under the initial load level. We selected this load level to ensure all experiments commence from a highly utilized and stable system state. The peak load for each benchmark is determined as the maximum query rate at which QoS targets are consistently met. The specific initial loads of HR, SN, OB, and EB1 are 6,000, 1,600, 1,500, and 3,000 queries per second.

Establishing this initial load allows us to subsequently vary the query mix ratios among different call graphs to simulate realistic dynamic scenarios. Thus, we adjust the loads of different numbers of services and call graphs to form various scenarios. For a benchmark with N_s services and N_g call graphs, we test all combinations of load changes from 0 to N_s services and from 0 to N_g call graphs. The load for each changed service or call graph is randomly increased or decreased by 30%. In Figures 12, 13, 16, 17, and 18, the x-axis shows the number of services and call graphs with load changes, e.g., *(3,7)* represents 3 services and 7 call graphs.

We compare Delphinus with two state-of-the-art systems ERMS [30] and Nodens [44]. ERMS mixed queries from different services and call graphs in the same containers of shared microservices. It does not identify queries from different call graphs and merges the call graphs to manage their resources uniformly. Except for the same mixed scheme, Nodens fine-grained monitors the load of each microservice for efficient resource allocation, but cannot identify queries from different services and call graphs. Moreover, ERMS and Nodens conduct only horizontal and vertical scaling, respectively. We optimize them to first vertical scale, and horizontal scale only when containers cannot support the loads. For fair comparisons, we set the same monitor interval (5 seconds) and the same container memory (100MB) for Delphinus and baselines.

We built an open-loop load generator to generate various loads for different services and call graphs, with the Gaussian distribution for the query inter-arrival times. Each load-generator thread produces a sequence of 250 queries per second based on this distribution, and we employ multiple concurrent threads for higher loads. We select this distribution since production microservice analysis found that inter-arrival patterns can be well approximated by a Gaussian distribution [29].

For each testing case, we apply a 1-minute workload and repeat 20 times per case, taking the average as the final result. Moreover, since popular academic benchmarks lack time-varying traces and diverse types of query mix ratios, we construct new benchmarks in Sections 8.9 and 8.10 directly from the Alibaba trace [31] to evaluate performance under production-like patterns. In detail, we evaluated Delphinus and baselines using a 24-hour trace under 4 different QoS target settings (Figures 19 and 20). Since each case required 24 hours to complete, no repetitions were conducted.

8.2 High Resource Efficiency under QoS Guarantee

We first evaluate Delphinus on resource efficiency while ensuring the QoS of diverse types of queries. We show typical test cases for the 4 benchmarks here and other cases show similar results.

Figure 12 shows the CPU and memory usage of Delphinus and baselines. As observed, Delphinus uses the least CPU cores (i.e., best vertical scale efficiency) in all cases, with an average reduction of 40.1% and 35.9% (22.5 and 20.1 cores) compared to ERMS and Nodens, respectively. Moreover, Delphinus reduces the memory usage (i.e., container number) in all cases, with an average reduction of 36.4% and 32.9% (616.3 MB and 557.3 MB) compared to ERMS and Nodens, respectively. It shows that Delphinus has higher horizontal scale efficiency.

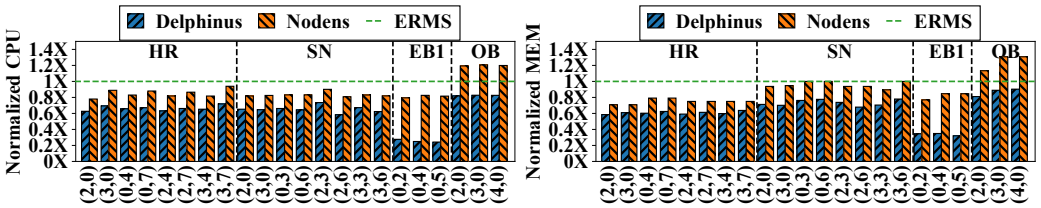


Fig. 12. The CPU and memory usage of benchmarks with Delphinus and Nodens normalized to ERMS.

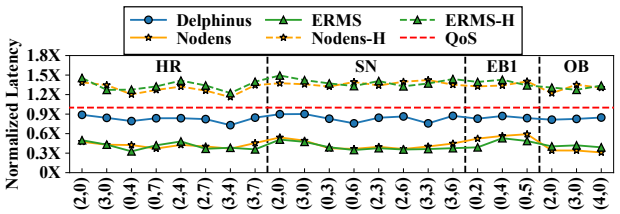


Fig. 13. The 99%-ile latency of benchmarks normalized to the QoS target with different systems.

The solid lines in Figure 13 show the 99%-ile latency under the resource allocation with Delphinus, ERMS, and Nodens, respectively, normalized to the QoS target of diverse queries. As shown, all systems can guarantee the latency QoS in all test cases. We can also observe that Delphinus has the 99%-ile latency that is closer to the QoS target. This is because Delphinus can more efficiently allocate computing resources to shared microservices than baselines while ensuring the QoS.

Delphinus can bring higher efficiency for the benchmarks with more shared microservices, more types of queries, and the resource usage of different types of queries showing higher diversity. For instance, *EB1* has better resource efficiency with Delphinus in Figure 12, because the resource usage of different types of queries of it can vary by 1.8× on average. Delphinus also shows superior efficiency for dominant sharing microservices. Taking the "ms-14" from EB1 as an example, Delphinus's CPU efficiency on it improves by 80.7% and memory efficiency improves by 81.1% compared to ERMS, while those values are 72.4% and 76.3% compared to Nodens.

As statistics of other test cases except the above typical ones, Delphinus also reduces CPU and memory usage by 37.3% and 33.9% on average under QoS compared to ERMS, and 33.4% and 30.8% compared to Nodens while ensuring the QoS. These results are consistent with the typical cases. As shown in Figure 14, Delphinus also improves the peak throughput by 1.7× and 1.5× on average with the same eight-node cluster compared to ERMS and Nodens. At the supported peak load of Delphinus, Delphinus ensures the QoS while ERMS and Nodens violate the QoS by 2.4× and 1.8×.

Moreover, to make a more apples-to-apples comparisons, we further allocate the same amount of resources determined by Delphinus to the two baselines. As ERMS-H and Noden-H in Figure 13, ERMS and Nodens result in 1.36× and 1.33× QoS violations, respectively. For each testing case in Figure 13, we also further selected the lowest latency achieved by Nodens and ERMS as the QoS for Delphinus and re-ran the experiments. Results show Delphinus still averagely uses 27.5% less CPU and 18.3% less memory compared to ERMS, and 22.2% and 13.8% compared to Nodens.

The above results are because, when diverse queries are served in a shared resource pool, it is hard for baselines to allocate resources dynamically to ensure each query type receives appropriate resources to meet its specific QoS. With the lack of fine-grained resource control, resources like

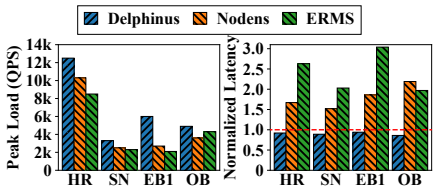


Fig. 14. The peak load, and the latency normalized to QoS under the peak load of Delphinus.

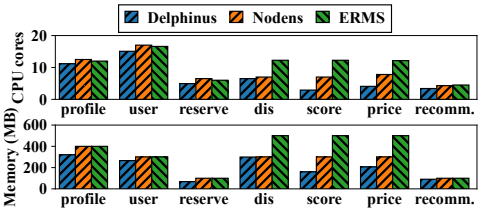


Fig. 15. An example resource usage in the test case *(2,4)* of the *HR*. The *recomm.* is short for *recommend*.

CPU cycles cannot be precisely allocated and isolated per query type. Thus, when allocating the tight resources determined by Delphinus to baselines, they cannot ensure the QoS of all types of queries, and thus require more resources to achieve the same latency performance.

At last, to evaluate Delphinus's effectiveness for dynamic memory usage of diverse types of queries, we also evaluated using an LR model to predict memory demand based on load like previous work [27] and performed vertical scaling accordingly for each container. Results show that Delphinus also reduces the memory usage by 28.7% and 20.6% compared to ERMS and Nodens.

8.3 Diving into the High Resource Efficiency

We use an example *(2,4)* of the *HR* to better explain Delphinus's advantages on shared microservices. Figure 15 shows the CPU and memory allocation to each shared microservice.

For *profile*, *user*, and *reserve* microservices that are shared among services, Delphinus has the least CPU and memory, as it identifies queries from services and schedules them to different container groups. Since ERMS mixed queries from different services, it uses more resources than Delphinus. Nodens not only fails to identify queries from services but also serves them together, hence showing the lowest efficiency. For *dis*, *score*, *price*, and *recommend* microservices that are shared among call graphs in a service, Delphinus also uses the least resources as it identifies queries for different call graphs and schedules the queries to different groups. Nodens uses more resources as it cannot identify different types of queries from call graphs. ERMS has the worst resource efficiency as it merges call graphs and allocates resources based on the total load of the corresponding service.

8.4 Effectiveness of the Query Type Identification

In this section, we show the performance of Delphinus-woq, a variant of Delphinus that disables query type identifications (Section 5.1). Since service or function grouping is dependent on the prior identification of different types of queries across services or functions, the group scheduling (Section 5.2) is also disabled, and diverse queries are served together in shared microservices.

The left part of Figure 16 shows the CPU and memory usage of all test cases with Delphinus-woq normalized to Delphinus's. As observed, the CPU and memory usage of Delphinus-woq are 2.9× and 3.0× on average compared to Delphinus's, respectively. Without query identifications, to ensure the QoS, each shared microservice is allocated based on the application's total load and the type of queries with the maximum resource demand. Moreover, horizontal scaling needs to be performed based on queries with the minimum single-container-supported load. We can also find that Delphinus-woq has worse resource efficiency for *HR* and *EB1*. This is because the shared microservices of them exhibit significant resource usage differences for different types of queries.

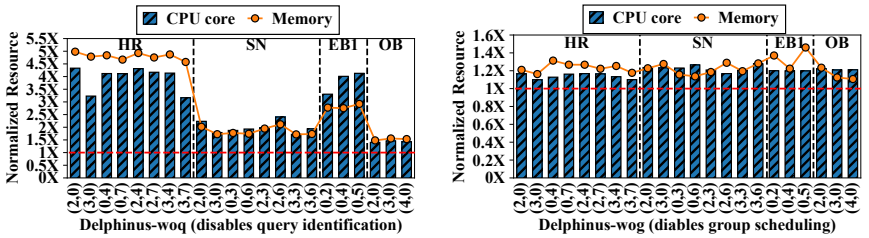


Fig. 16. The resource usage of Delphinus-woq and Delphinus-wog normalized to Delphinus, respectively.

8.5 Effectiveness of the Group Scheduling

In this subsection, we show the performance of Delphinus-wog, a variant of Delphinus that disables the group scheduling of queries (Section 5.2). Although different types of queries can be identified, they are served together in the same containers. This makes the resource allocation based on Section 6.1 fail to guarantee the QoS. Thus, we use brute-force profiling in Delphinus-wog to find the minimal required resources for each shared microservice.

The right part of Figure 16 shows the CPU and memory usage of Delphinus-wog in all test cases normalized to Delphinus. We can observe that Delphinus-wog requires 1.2× and 1.2× CPU and memory resources on average, respectively, compared to Delphinus. Without the grouped scheduling, different types of queries are served together in the same containers of shared microservices, which needs more CPU cores and containers to ensure the QoS of all query types. Moreover, we can observe that Delphinus-wog performs better than Delphinus-woq on the resource allocation. This is because Delphinus-wog identifies diverse types of queries from services and call graphs, which is the basis of grouped scheduling and efficient resource allocation.

We also further evaluate the latency of Delphinus-wog when allocating the same amount of resources as Delphinus to it. Results show Delphinus-wog suffers an average QoS violation of 2.4×.

8.6 Effectiveness of the Service Identification and Group Scheduling

In this subsection, we show the performance of Delphinus-wos, a variant of Delphinus that disables the query type identification only at the service level. With Delphinus-wos, the queries to diverse services cannot be identified and these queries are served together in the same containers, but queries from diverse RPC functions can be identified and scheduled to different container groups.

The left part of Figure 17 shows that Delphinus-wos uses more CPU and memory in all test cases, with an average increase of 2.2× and 2.4× compared to Delphinus, respectively. When not identifying queries from different services, the microservices shared by different services have low resource efficiency. The reasons lie in the fact that these microservices have to be allocated CPU cores based on the service with the maximum demand, while horizontal scaling should be conducted based on the minimum single-container-supported load. For some special cases, Delphinus-wos in *EB1* equals as Delphinus, since *EB1* only has a single service and service identification does not work. Moreover, Delphinus-wos's resource usage is the same as Delphinus-woq's for the *OB*. This is because the services in *OB* do not have multiple call graphs with diverse RPC functions, thus even though Delphinus-woq further disables RPC function identification, it does not make a difference.

We also exploit the synergism between mechanisms with the resource usage increase breakdown of: ① service type identification, and ② group scheduling according to service types. On average, ① and ② increase CPU usage by 2.05× and 1.12×, contributing to 89.7% and 10.3% of the total

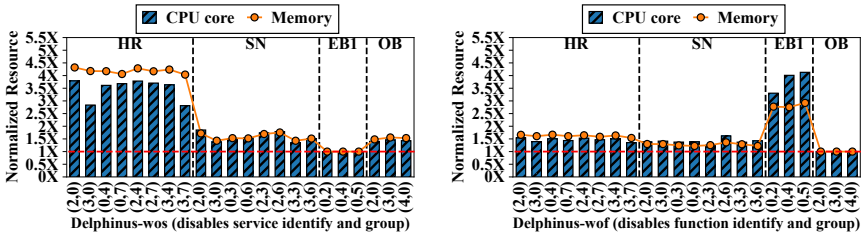


Fig. 17. The resource usage of Delphinus-wos and Delphinus-wof normalized to Delphinus, respectively.

increase, respectively. For memory, they increase by 2.29× and 1.15×, i.e., contributing 89.6% and 10.4%, respectively. These results confirm that both components are essential, with identification being the primary contributor as it is the basis of group scheduling.

8.7 Effectiveness of the RPC Function Identification and Group Scheduling

In this subsection, we show the performance of Delphinus-wof, a variant of Delphinus that disables the query identification only at the RPC function level. With Delphinus-wof, the queries to diverse RPC functions cannot be identified and these queries are served together in the same containers.

The right part of Figure 17 shows the CPU and memory usage with Delphinus-wof normalized to Delphinus. As observed, Delphinus-wof has larger CPU and memory usage in all the test cases, which is 1.7× and 1.6× on average of Delphinus's, respectively. When not identifying various RPC functions, the microservices shared by queries across call graphs within the same service still have resource inefficiency. This is because the CPU cores of these microservices can only be allocated based on the RPC function with the maximum demand, and the horizontal scaling needs to be performed according to the minimum single-container-supported load. Delphinus-wof's resource usage in *EB1* is equal to Delphinus-woq. This is because *EB1* only has one service, and there are no differences in whether service identifications exist. Moreover, Delphinus-wof equals Delphinus in *OB*, as the services in *OB* do not have multiple call graphs.

We also break down the resource usage increase of: ① RPC function type identification, and ② group scheduling according to RPC function types. Results show that ① and ② averagely increase the CPU usage by 1.61× and 1.08×, i.e., contributing to 88.4% and 11.6% of the total increase, respectively. For memory, they increase by 1.47× and 1.10×, i.e., contributing to 82.5% and 17.5%, respectively. These results demonstrate that both components are crucial to resource efficiency.

8.8 Effectiveness of the Container Borrowing

Since separate groups may lead to long tail latency after loads change, we evaluate the in-process tail latency of Delphinus-wob, a variant of Delphinus that disables the container borrowing mechanism. The in-process tail latency is the 99%-ile latency of queries during the scaling. We also investigate the QoS recovery time, which represents the time needed to recover the QoS after dynamics [44].

Figure 18 shows the in-process tail latency and QoS recovery time of Delphinus-wob normalized to Delphinus. Compared to Delphinus, the in-process tail latency and recovery time are increased in all test cases, with an average increase of 4.0× and 2.5×, respectively. When the container borrowing mechanism is disabled, lots of queries are queued during horizontal scaling due to imbalanced resource usage across container groups. These results demonstrate that Delphinus solves the problem of imbalanced resource utilization across isolated groups when the loads change.

Moreover, we break down the performance decreases of Delphinus-wob. Corresponding to Figure 11, the tail latency and QoS recovery time respectively increase to 1.7× and 1.8× without ①, 1.9× and 1.3× without ②, and 2.4× and 1.4× without ③, on average compared to Delphinus. This

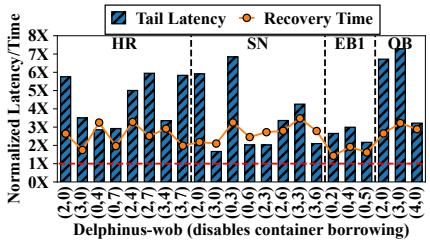
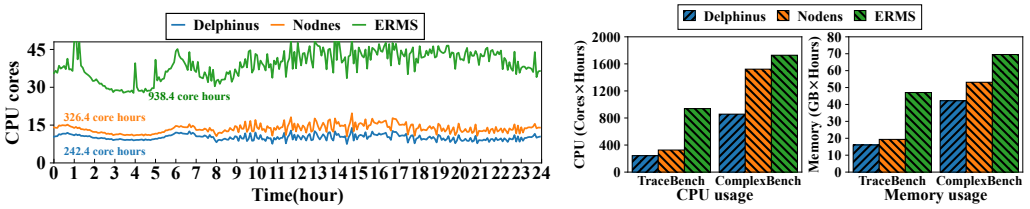


Fig. 18. The tail latency and recovery time of Delphinus-wob normalized to Delphinus, respectively.



(a) The CPU core usage of the *TraceBench*.

(b) Statistical CPU and memory usage.

Fig. 19. The resource usage of Delphinus and baselines on *TraceBench* and *ComplexBench* with one-day traces.

breakdown confirms that no single mechanism is redundant, each addresses a distinct performance bottleneck.

8.9 Evaluations Using Production Traces

In this subsection, we select the service with shared microservices that has the largest number of queries in the Alibaba microservice trace [28, 31], which has 8 microservices, 8 call graphs, and 3 microservices shared among call graphs in this service. According to its call graphs, we construct a benchmark (*TraceBench*) by using gRPC [18], and adopt commonly-used workloads (e.g., Page Rank and Word Stemming) for it. The above constructions are the same to prior microservice scheduling studies [33, 34, 44]. We evaluate this benchmark with a one-day trace. The load on this day is lower during 0h-8h and higher during 8h-24h, averaging 515.1 and 2119.4 queries per minute, respectively.

Figure 19(a) shows the CPU core usage of Delphinus, ERMS, and Nodens evaluated on *TraceBench* with the one-day traces. Under the QoS (300ms) of all systems during the 24 hours, Delphinus reduces the CPU core usage in both low-load and high-load periods. As shown in Figure 19(b), the total CPU core hours during this day of Delphinus/ERMS/Nodnes are 242.4/938.4/326.4, thus Delphinus reduces the CPU usage by 74.2%/25.7% compared to ERMS/Nodens. Memory usage shows a similar trend to CPU, as statistics in Figure 19(b), Delphinus reduces memory usage by 65.9% and 17.1% compared to ERMS and Nodens, respectively. The memory efficiency is slightly lower than CPU, as we remain at least one container for each group, which may waste some memory at low loads.

The baselines serve the queries of shared microservices in a shared resource pool, which cannot allocate different types of queries with fine-grained resources. ERMS has the worst resource efficiency, as this benchmark constructed from traces only has microservices shared among call graphs. ERMS merges all call graphs to allocate resources for each microservice shared across call graphs with the total service load. Therefore, it also has worse resource usage results compared to benchmarks in Section 8.2 (with microservices both shared by services and call graphs).

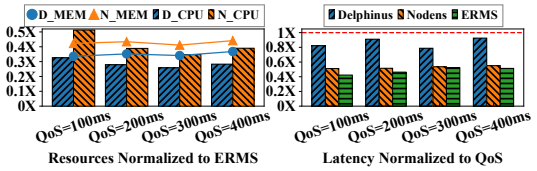


Fig. 20. The resource usage and 99%-ile latency for evaluations of *TraceBench* on one-day traces under various QoS targets. The *D* and *N* are short for Delphinus and Nodens, respectively.

Moreover, to investigate even more complex topologies, we merged the top two services by query volume from the Alibaba trace [31] to generate the *ComplexBench*. It consists of 2 services, 9 call graphs, 25 microservices, 9 microservices shared across different call graphs, and 2 shared across different services. Using the same 24-hour trace, as the statistics shown in the Figure 19(b), Delphinus reduces CPU usage by 50.4% and memory usage by 39.2% compared to ERMS, and the values are 43.7% and 20.6% compared to Nodens, while ensuring the QoS. These results demonstrate that Delphinus also has better efficiency with more complex production-level microservice call graphs.

8.10 Handling Different Latency QoS Targets

In this subsection, we evaluate Delphinus's effectiveness on QoS and resource efficiency under different latency QoS targets, by using the *TraceBench* on the same 24-hour traces as Section 8.9.

The left part of Figure 20 compares resource usage between Delphinus and baselines under different QoS targets. Delphinus reduces CPU usage by 67.4%–74.2% and memory by 63.2%–66.4% compared to ERMS, and by 25.7%–36.4% and 16.6%–20.5% compared to Nodens. The right part shows the 99%-ile latency under different QoS targets. All systems ensure the QoS, but Delphinus consistently stays closer to the QoS due to its higher resource efficiency.

Changing QoSs inherently alters resource demands for each type of queries at the same load. Delphinus can adaptively fit load-to-resource prediction models for each type of queries of shared microservices under different QoS targets. It identifies each type of queries and schedules them to separate container groups, enabling minimal resource allocation at any given QoS. By contrast, ERMS and Nodens mix query serving at shared microservices and have to over-provision resources based on the highest-demand query type to ensure QoS. This is why they stay far below QoS, while Delphinus stays close. Thus, Delphinus ensures any QoS with similar efficiency gains.

8.11 Overhead of Delphinus

Delphinus's overhead can be classified into per-query overhead and resource re-scaling overhead.

For the scheduling of each query, the operation to retrieve the service ID from the RPC header and perform the routing lookup is highly optimized, which processes only 18 bytes of metadata and introduces a low additional latency of 0.18μs on average. In total, the decision of the query identification and scheduling can be done within 0.15 ms in the query scheduler, which has a similar overhead to the default Round-Robin strategy. For the resource re-scaling that operates every 5 seconds, the time overhead is about 7 ms to get the loads of each RPC function in each service from query identifications, about 5ms for scaling decisions and enabling container borrowing in the load adapter, and less than 10ms for updating the container numbers for shared microservice groups.

For the horizontal scaling, we leverage standard Kubernetes [23], whose overhead is a well-studied platform-level issue, and solutions like lightweight container [2, 26] are orthogonal to our work. Moreover, Delphinus's container-borrowing mechanism directly mitigates this overhead by

redistributing idle containers to overloaded ones. In our evaluations, the average startup time for a new container is about 230 ms. For the vertical scaling of re-configuring resources, we directly utilize cgroups [8] without pod recreation. Each update completes in less than 1ms in our evaluations.

The above overheads are acceptable since (1) the added latency for query routing and scheduling (about 0.15 ms) is orders of magnitude smaller than the typical execution time of a query (hundreds of milliseconds). (2) The resource re-allocation runs every 5 seconds. The time to collect data and make a scaling decision is only 22 ms in total, which is insignificant relative to this time interval.

9 Conclusion

In this article, we propose Delphinus to enable efficient resource management of shared microservices while ensuring the QoS of diverse queries. Delphinus's query scheduler introduces a two-level scheme to identify queries with diverse resource demands according to their service and RPC function IDs. Based on this scheme, diverse queries are scheduled to separate container groups of shared microservices. Delphinus's load adapter efficiently scales resources for the shared microservices and fully utilizes idle containers among shared microservice groups when loads of diverse queries change. We have implemented Delphinus and evaluation results show that Delphinus reduces the CPU and memory resources by 40.1% and 36.4%, respectively, compared to state-of-the-art works.

References

[1] HTTP2: Create additional connections when maximum active streams is reached. 2020. Retrieved May 4, 2026 from https://github.com/dotnet/runtime/issues/35088

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation*. 419–434.

[3] Azure Container Apps. 2025. Retrieved May 4, 2026 from https://azure.microsoft.com/en-us/products/

[4] Atomicity and Transactions. 2026. Retrieved May 4, 2026 from https://www.mongodb.com/docs/manual/core/write-operations-atomicity/

[5] Ataollah Fatahi Baarzi and George Kesidis. 2021. Showar: Right-sizing and efficient scheduling of microservices. In *Proceedings of the ACM Symposium on Cloud Computing*. 427–441.

[6] Load Balancing. 2026. Retrieved May 4, 2026 from https://deepwiki.com/grpc/grpc-go/4-load-balancing

[7] Online Boutique. 2025. Retrieved May 4, 2026 from https://github.com/GoogleCloudPlatform/microservices-demo/

[8] cgroups(7) – Linux manual page. 2024. Retrieved May 4, 2026 from https://man7.org/linux/man-pages/man7/cgroups.7.html

[9] Jinyuan Chen, Jiuchen Shi, Quan Chen, Lin Gu, and Minyi Guo. 2026. Veyth: Adaptive container placement for optimizing cross-server network traffic of microservice applications. In *Proceedings of the Advanced Parallel Processing Technologies*. 201–211.

[10] 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 *Proceedings of the 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture*. 424–436.

[11] Ka-Ho Chow, Umesh Deshpande, Sangeetha Seshadri, and Ling Liu. 2022. DeepRest: Deep resource estimation for interactive microservices. In *Proceedings of the 17th European Conference on Computer Systems*. 181–198.

[12] 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 *Proceedings of the 39th ACM International Conference on Supercomputing*. 895–910.

[13] Kaihua Fu, Wei Zhang, Quan Chen, Deze Zeng, and Minyi Guo. 2021. Adaptive resource efficient microservice deployment in cloud-edge continuum. *IEEE Transactions on Parallel and Distributed Systems* 33, 8 (2021), 1825–1840.

[14] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 2021. Sage: Practical and scalable ML-driven performance debugging in microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 135–151.

[15] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. 3–18.

[16] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. 19–33.

[17] Alim Ul Gias, Giuliano Casale, and Murray Woodside. 2019. ATOM: Model-driven autoscaling for microservices. In *Proceedings of the 2019 IEEE 39th International Conference on Distributed Computing Systems*. 1994–2004.

[18] gRPC. 2026. Retrieved May 4, 2026 from https://grpc.io/

[19] Md Rajib Hossen, Mohammad A. Islam, and Kishwar Ahmed. 2022. Practical efficient microservice autoscaling with QoS assurance. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*. 240–252.

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

[21] Jaeger. 2025. Retrieved May 4, 2026 from https://www.jaegertracing.io/

[22] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. 2019. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. In *Proceedings of the 14th EuroSys Conference 2019*. 1–16.

[23] Kubernetes. 2025. Retrieved May 4, 2026 from kubernetes.io

[24] Ye Li, Jian Tan, Bin Wu, Xiao He, and Feifei Li. 2023. ShapleyIQ: Influence quantification by shapley values for performance debugging of microservices. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 287–323.

[25] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. 2022. RunD: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In *Proceedings of the 2022 USENIX Annual Technical Conference*. 53–68.

[26] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, et al. 2022. Help rather than recycle: Alleviating cold startup in serverless computing through inter-function container sharing. In *Proceedings of the 2022 USENIX Annual Technical Conference*. 69–84.

[27] 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 Transactions on Computer Systems* 42, 1–2 (2024), 1–28.

[28] 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*. 412–426.

[29] 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.

[30] 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*. 62–77.

[31] 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*. 355–369.

[32] Is memcached atomic? 2026. Retrieved May 4, 2026 from https://docs.memcached.org/userguide/faq/#is-memcached-atomic

[33] Amirhossein Mirhosseini, Sameh Elnikety, and Thomas F Wenisch. 2021. Parslo: A gradient descent-based approach for near-optimal partial SLO allotment in microservices. In *Proceedings of the ACM Symposium on Cloud Computing*. 442–457.

[34] Amirhossein Mirhosseini, Brendan L. West, Geoffrey W. Blake, and Thomas F. Wenisch. 2020. Q-Zilla: A scheduling framework and core microarchitecture for tail-tolerant microservices. In *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture*. 207–219.

[35] Microservices Engine (MSE). 2025. Retrieved May 4, 2026 from https://www.alibabacloud.com/product/microservices-engine

[36] Limiting number of pendingStreams. 2026. Retrieved May 4, 2026 from https://groups.google.com/g/grpc-io/c/9IbBfoOezJQ

[37] Custom Load Balancing Policies. 2026. Retrieved May 4, 2026 from https://grpc.io/docs/guides/custom-load-balancing/

[38] 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 Symposium on Operating Systems Design and Implementation*. 805–825.

[39] Concurrent request/connection limits for Go servers. 2026. Retrieved May 4, 2026 from https://pkg.go.dev/github.com/evanj/concurrentlimit

[40] Korakit Seemakhupt, Brent E. Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C. Snoeren, Arvind Krishnamurthy, David E. Culler, and Henry M. Levy. 2023. A cloud-scale characterization of remote procedure calls. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 498–514.

[41] 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.

[42] 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 Transactions on Parallel and Distributed Systems* 35, 9 (2024), 1565–1582.

[43] 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 *Proceedings of the 2022 IEEE International Parallel and Distributed Processing Symposium*. 324–334.

[44] 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 *Proceedings of the 2023 USENIX Annual Technical Conference*. 403–417.

[45] Akshitha Sriraman and Thomas F. Wenisch. 2018. μ suite: A benchmark suite for microservices. In *Proceedings of the 2018 IEEE International Symposium on Workload Characterization*. 1–12.

[46] 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.

[47] Prometheus Monitoring system and time series database. 2025. Retrieved May 4, 2026 from https://prometheus.io/

[48] tc(8) – Linux manual page. 2025. Retrieved May 4, 2026 from https://man7.org/linux/man-pages/man8/tc.8.html

[49] Introduction to microservices. 2021. Retrieved May 4, 2026 from https://cloud.google.com/architecture/microservices-architecture-introduction

[50] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the curtains of serverless platforms. In *Proceedings of the 2018 USENIX Annual Technical Conference*. 133–146.

[51] Zibo Wang, Pinghe Li, Chieh-Jan Mike Liang, Feng Wu, and Francis Y. Yan. 2024. Autothrottle: A practical Bi-level approach to resource management for SLO-targeted microservices. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation*. 149–165.

[52] Wei Zhang, Quan Chen, Kaihua Fu, Ningxin Zheng, Zhiyi Huang, Jingwen Leng, and Minyi Guo. 2022. Astraea: towards QoS-aware and resource-efficient multi-stage GPU services. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 570–582.

[53] 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.

[54] Yanqi Zhang, Zhuangzhuang Zhou, Sameh Elnikety, and Christina Delimitrou. 2024. Ursa: Lightweight resource management for cloud-native microservices. In *Proceedings of the 2024 IEEE International Symposium on High-Performance Computer Architecture*. 954–969.

[55] Laiping Zhao, Yanan Yang, Kaixuan Zhang, Xiaobo Zhou, Tie Qiu, Keqiu Li, and Yungang Bao. 2020. Rhythm: Component-distinguishable workload deployment in datacenters. In *Proceedings of the 15th European Conference on Computer Systems*. 1–17.

[56] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2018. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering* 47, 2 (2018), 243–260.

[57] Xianzhi Zhu, Yongkun Li, Lulu Yao, Zhihao Qi, Yinlong Xu, Pengcheng Wang, Weiguang Wang, and Xia Zhu. 2023. On optimizing traffic scheduling for multi-replica containerized microservices. In *Proceedings of the 52nd International Conference on Parallel Processing*. 358–368.

Received 20 August 2025; revised 5 March 2026; accepted 21 April 2026