Nodens: Enabling Resource Efficient and Fast QoS Recovery of Dynamic Microservice Applications in Datacenters

Jiuchen Shi, Hang Zhang, Zhixin Tong, Quan Chen, Kaihua Fu, and Minyi Guo,
Department of Computer Science and Engineering, Shanghai Jiao Tong University

https://www.usenix.org/conference/atc23/presentation/shi

This paper is included in the Proceedings of the 2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

Nodens: Enabling Resource Efficient and Fast QoS Recovery of Dynamic Microservice Applications in Datacenters

Jiuchen Shi, Hang Zhang, Zhixin Tong, Quan Chen, Kaihua Fu, Minyi Guo
Department of Computer Science and Engineering, Shanghai Jiao Tong University

Abstract

Current microservice applications always meet with load and call graph dynamics. These dynamics can easily lead to inappropriate resource allocation for microservices, and further lead to Quality-of-Service (QoS) violations of applications. However, current microservice management works are incapable to handle these dynamics, mainly due to the execution blocking effect among microservices. We therefore propose Nodens, a runtime system that enables fast QoS recovery of the dynamic microservice application, while maintaining the efficiency of the resource usage. Nodens comprises a *traffic-based load monitor*, a *blocking-aware load updater*, and a *resource-efficient query drainer*. The load monitor periodically checks microservices' network bandwidth usage and predicts the monitored loads based on it. The load updater updates the actual "to-be-processed" load of each microservice to enable fast resource adjustment. The query drainer allocates just-enough excessive resources for microservices to drain the queued queries, which can ensure the QoS recovery time target. Our experiments show that Nodens can reduce the QoS recovery time by 12.1X with only the excessive resource usage of 6.1% on average, compared to the state-of-the-art microservice management systems.

1 Introduction

User-facing applications are evolving towards the microservice architecture, with which the microservices communicate through the network [25, 41] and are able to scale independently [1, 5, 9]. The dependencies of the microservices can often be denoted by a Directed Acyclic Graph (DAG) [31, 32], each node represents a microservice and each edge represents the call dependency [24, 38]. Moreover, a production microservice application often has multiple call graphs [31, 32], as users have different query patterns. Figure 1 shows an example dependency graph and two call graphs that handle different user queries.

In these applications, the load of each microservice change dynamically, because 1) the load of the entire application may

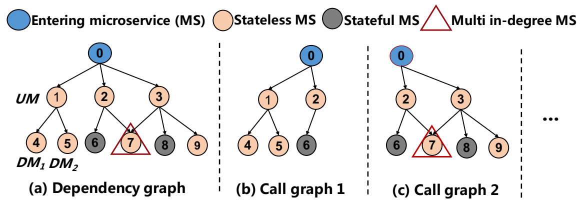


Figure 1: An example microservice dependency graph and two call graphs.

change over time due to the diurnal, irregular, and bursty load patterns (referred to be "load dynamic") [3, 19, 28, 40], and 2) the percentages of queries that go to different call graphs may change over time (referred to be "call graph dynamic") [34]. We analyze the open-sourced production traces [2, 8], and find that the load variation is 30% on average (up to 7.5X), and the percentages of queries that go to different call graphs also vary by 15% on average (up to 70%).

While it is crucial to ensure the required Quality-of-Service (QoS) of user-facing applications [20, 30, 39], prior works [22, 38, 41, 44] fail to handle these dynamic applications. Most prior works periodically check the load of each microservice, and adjust the resource allocation of each microservice based on the monitored load. They are incapable for the current dynamic microservice applications for two main reasons.

As for the first reason, the monitored load of a microservice may not be its real "to-be-processed" load due to the cascade call relationship. Many queries may be blocked by its upstream microservices. A microservice may not be allocated enough computation resources in this case. Worse, there is a lag in noticing the load increase. For instance, if the monitoring period is 1 second, the lag can be 1 second as well. However, the short lag may result in the long QoS violation, as a great many of queries may queue up at the microservice. A very long time is needed to adjust resources and drain up the query queue, when the microservice's resource is allocated based on the monitored load. Our experiments show that the QoS can be recovered in as long as 84.4 seconds.

Some other prior work [18, 23, 25, 45] predict the QoS and adjust the resource allocation beforehand. They assume all the queries go through all the microservices, thus are not able to handle the dynamic loads due to the variation of call graphs.

An intuitive solution is calculating the actual "to-be-processed" queries of each microservice based on the dependency graph and the call graphs, and adjusting their resources accordingly. However, it does not work because we find that the queries of a microservice may also be blocked by other microservices besides of its upstream microservices in the dependency graph. For instance, microservice-2 in Figure 1 may be blocked by microservice-4 or microservice-5 that do not call it in the dependency graph. This happens when microservice-0 calls microservice-1 and microservice-2 in a fixed order and the resource allocation of microservice-4 or microservice-5 is insufficient.

An appropriate solution should be able to capture all the potential "blocking" relationships, and be able to drain up the query queues due to the monitoring lag. We therefore define an *execution blocking graph* that captures all the superior microservices that may block a microservice, based on which we further propose a runtime system named **Nodens**[1] that enables fast QoS recovery, if the loads of some microservices suddenly increase due to the two types of dynamics. Note that the execution blocking graph is not the same as the microservice dependency graph.

Nodens comprises a *traffic-based load monitor*, a *blocking-aware load updater*, and a *resource-efficient query drainer*. For each microservice, the load monitor periodically checks the input network traffics, and predicts the current monitored load of the microservice based on the traffics. This method is much faster than obtaining the load information from the microservices' interfaces, enabling earlier resource allocation adjustment. The load updater updates the execution blocking graph with the monitored load obtained from the monitor, and estimates the actual "to-be-processed" load of each microservice. The query drainer adjusts the CPU resource allocated to each microservice based on the actual loads of the microservices and the queued queries during the previous process, in order to quickly recover the QoS.

This paper makes three main contributions.

- **Comprehensive analysis of current methods to handle microservice dynamics.** The insights obtained from the analysis identify the opportunities to enable fast QoS recovery when dealing with microservice dynamics.

- **The design of a method to update actual loads of microservices under execution blocking effect.** We construct the execution blocking graph based on microservice dependencies, with which we can update actual loads of microservices under the blocking effect.

- **The design of a policy to drain the queued queries during the resource adjustment process.** The policy allocates excessive resources for microservices to satisfy the QoS recovery time target, while maintaining the resource efficiency.

[1]The source code is available at https://github.com/shijiuchen/Nodens.

We evaluate Nodens with our benchmarks on an eight-node cluster. The experimental results show that Nodens can reduce the QoS recovery time by 12.1X with only the over-provisioned resources of 6.1% on average, compared to the state-of-the-art microservice management systems.

2 Related Work

There has been some related work on ensuring the QoS of user-facing applications.

2.1 Reactive Microservice Management

Reactive microservice management systems periodically monitor the state (e.g., load or latency) of each microservice, and adjust the resource allocation of each microservice based on the state.

Heuristic methods: SHOWAR [17], PEMA [27], Astraea [44], and ATOM [26] designed heuristic approaches to conduct horizontal or vertical scaling for CPU or GPU microservices based on the resource utilization and response latency. These heuristic methods can determine the resource allocation for microservices in a quick way, but hard to achieve the near-optimal values.

Machine Learning (ML) based methods: Nautilus [22] used Reinforcement learning (RL) as feedback to tune microservices' resources based on the application's response latency. FIRM [38] identified the critical microservices which caused QoS violations, and used RL as feedback to adjust resources for each microservice based on tail latencies, to guarantee the QoS of the application. ELIS [41] utilized the bayesian optimization algorithm with tail latencies as input to recycle the over-provisioned resources and then allocate just-enough resources to critical microservices. These ML-based methods can allocate near-optimal resources for microservices, but are slower due to incremental training under microservice dynamics.

Moreover, above reactive methods all have long QoS recovery time when handling dynamics of microservices, which is caused by the long monitoring interval, and the load blocking under the cascade call relationship among microservices.

2.2 Proactive Microservice Management

Proactive microservice management systems predicted the performance and resource allocation for microservices based on historical data. Seer [25] and Sage [23] used ML-based methods to predict the microservices that cause QoS violations based on the latency metrics, and increase the allocated resources for them. Sinan [45] and DeepRest [18] utilized

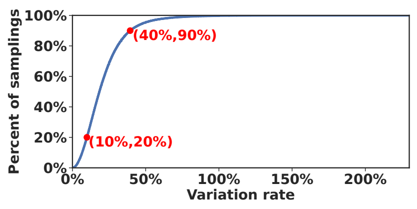


Figure 2: Load variation of top 20 microservice applications.

the deep learning-driven methods to predict the end-to-end latency and estimate the resource allocation for different microservice stages, which can minimize the resource usage while ensuring the QoS target. These works only consider a unique dependency graph, which cannot handle the call graph dynamics of the microservice application. Moreover, Madu [34] predicted the load size for microservices based on time series prediction models with the consideration of dynamic call graphs into the loss function. However, it cannot handle the unpredictable dynamic load and call graph cases that commonly exist in production traces [2, 40].

3 Investigating Dynamic MS Applications

In this section, we first analyze the production trace in the current public cloud. Then, we introduce microservice benchmarks we make which have the dynamic call graph features. At last, we explore the challenges of the current microservice management systems in dealing with dynamics.

3.1 Dynamic Loads and Call Graphs

We analyze the open-sourced production-level microservice trace [2] that contains the microservice call dependencies across 3000+ applications in 12 hours to show the dynamics.

In the analysis, we record the loads of the microservices for every five seconds, and calculate the load variation. The load variation is defined to be the load changes in the adjacent samples. Figure 2 shows the cumulative distribution of the load variation of the microservices in the top 20 production-level microservice applications. The top-20 applications are selected according to the numbers of their queries. As observed, the load variation is from 10% to 40% for 70% of the samples. In the worst case, the load may increase by 2.3X.

From statistics, we find some of the top 20 microservice applications have a great many types of call graphs, with a maximum of 53761 types. All the microservices touched by a query form a call graph. Different queries may have the same call graph. Moreover, Figure 3 shows the call graph proportion variation over time of the top 5 call graphs in the largest microservice application. A call graph's proportion variation is defined to be its proportion changes in the adjacent samples. We can observe that the percentages of queries to the call graphs change dynamically with no oblivious pattern.

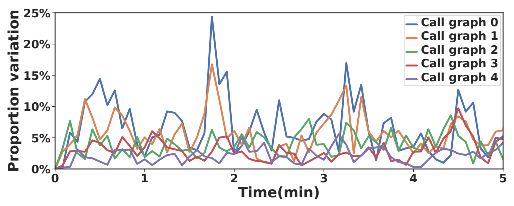


Figure 3: The call graph proportion variation of the largest microservice application in Alibaba Cloud.

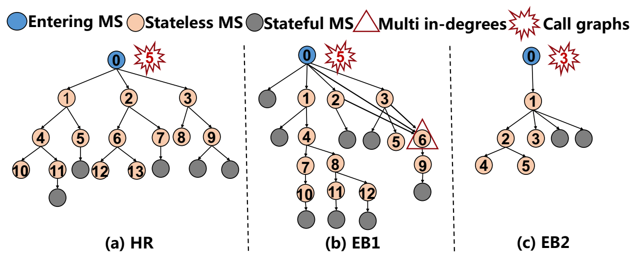


Figure 4: The dependency graphs of the benchmarks. Stateful microservices includes memcached and monogdb.

With the large number of call graphs and the unstable variation, it is non-trivial to profile the call graphs and their patterns, and define static optimal resource allocation beforehand.

3.2 The Investigation Benchmarks

While current microservice benchmarks do not support dynamic call graphs [24, 42, 46], and the traces [2] do not include the actual microservices (only the call traces), we build three benchmarks by integrating the call graph patterns of the trace and actual microservices in the benchmark suites.

Figure 4 shows the dependency graphs of the three benchmarks, and the number of call graphs. The *HR* benchmark is revised based on the popular HotelReservation benchmark [24]. It has five call graphs that respond to five types of user queries: nearest hotel search, highest rated hotel search, cheapest hotel recommend, comprehensive hotel recommend, and hotel reservation. The benchmarks *EB1* and *EB2* are created based on the dependency graphs of the top 2 applications with multiple call graphs in the trace. For *EB1* and *EB2*, similar to current benchmarks and related work [24, 35, 36, 46], we use commonly-used workloads in microservices, i.e., Nearest Neighbor Searching [6], Word Stemming [13], Quick Sort [15], Float Calculation [43], and Page Rank [16] to be the stateless microservices.

3.3 The Long QoS Recovery Time

We show the QoS recovery time and 99%-ile latencies of the benchmarks with load and the call graph dynamics in this subsection. The QoS recovery time is the time needed to reduce the 99%-ile latency to be below a fixed latency target

Table 1: Experiment specifications

	Specifications
Hardware	Eight-node cluster, Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz, 256GB Memory Capacity, 25 MiB L3 Cache Size (20-way set associative)
Software	Ubuntu 20.04.2 LTS with kernel 5.11.0-34-generic Docker version 20.10.18, Kubernetes version v1.20.4

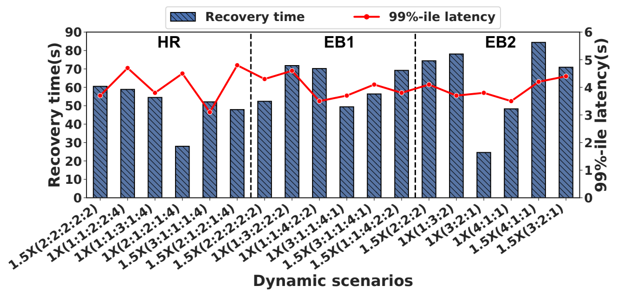


Figure 5: The QoS recovery time and 99%-ile latencies of benchmarks with ELIS.

(e.g., 100ms) after microservice dynamics happen. Table 1 summarizes the detailed hardware and software configurations. ELIS [41] uses bayesian optimization to tune resource allocation for microservices. We use ELIS as the representative resource management system for microservices in this section. Other systems have similar results and we show them in the evaluation section.

In the experiments, each benchmark has 6 dynamic load and call graph scenarios. We run the experiments on three identical servers managed with Kubernetes [14]. We expand the evaluation on eight servers in Section 8. In each test, each microservice is allocated the hand-tuned enough resource, and the numbers of queries to different call graphs are the same. Figure 5 shows the QoS recovery time and 99%-ile latencies in all the test cases. The x-axis represents the dynamic scenarios. For instance, *1.5X(2:1:2:1:4)* means the load increases to 1.5X, and the proportions of the 5 call graphs change to 2/10, 1/10, 2/10, 1/10, and 4/10. As observed, the QoS recovery time ranges from 24.6 to 84.4 seconds, and the 99%-ile latencies range from 3.1 to 4.8 seconds, in all the test cases.

Both the two types of dynamics result in serious QoS violations and the QoS recovery time is long.

3.4 Causes of The Long Recovery Time

Our investigation shows the long QoS recovery time is caused by *long monitoring interval*, *execution blocking effect due to dynamcis*, and *slow query draining*.

3.4.1 Long Monitoring Interval

Current resource management systems monitor the realtime latencies of the microservices, and reallocate resources based on either heuristic methods [17, 27, 44] or machine learning

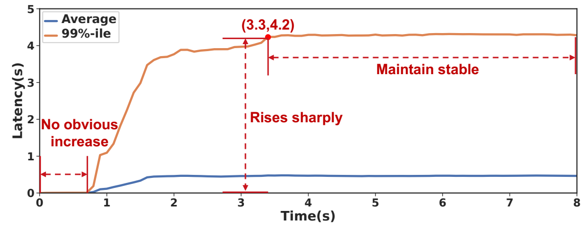


Figure 6: The exploration of latency monitoring intervals.

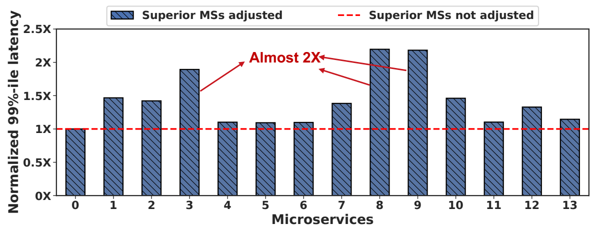


Figure 7: The 99%-ile latency of each microservice after its superior microservices have been allocated enough resources.

based methods [23, 38, 45]. These systems often use seconds or tens of seconds to be the monitoring interval.

As an example, Figure 6 shows the 99%-ile latency and average latency of the benchmark *HR*, when we increase its load to 1.5X. The x-axis shows the time since we increase the load. As observed, the monitored 99%-ile latency has no obvious increase in the first second, even if the load already increases. The 99%-ile latency starts to increase sharply at about the first second, and becomes stable after 3.3 seconds. This is because the latencies of the newly arrived queries are not reported before they complete. In this case, the 99%-ile latency reported in the first second is actually the latency before the load actually increases.

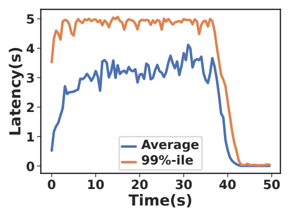
Long monitoring intervals are required for current systems that rely on the latencies of the microservices to adjust the resources. However, a great many of queries may already queue up at a microservice during the long monitoring interval.

3.4.2 Execution Blocking Effect

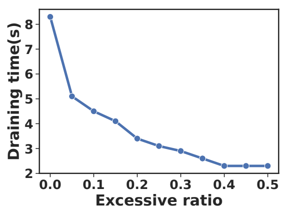
The second problem is that the monitored realtime load of a microservice may not be its actual "to-be-processed" load.

For instance, as shown in Figure 4(a), the load of microservice-3 may be blocked by microservice-0 if microservice-0 does not have enough computation resources. Similarly, the loads of microservice-8 and microservice-9 may also be blocked by microservice-3. There is more complex blocking effect, besides of the simple dependency relationship. The effect is referred to be *execution blocking effect* in this paper. We will analyze the effect in detail in Section 6.

Figure 7 shows the latencies of the microservices in the *HR* benchmark, when we allocated their superior microservices enough computation resources, normalized to their performance with the default resource allocation. The dynamic



(a) QoS recovery process.



(b) Excessive resource ratio.

Figure 8: The results of queued query draining.

scenario of this experiment is *1.5X(2:1:2:1:4)*. As observed, the latencies of many microservices increase when their superior microservices get enough resources. This is because these microservices may get the real "to-be-processed" loads when the blocking effect is alleviated.

Due to the execution blocking effect, the monitored load of a microservice may be much smaller than its actual "to-be-processed" load. Current methods based on monitored load may not allocate enough resources for microservices, thus require to adjust the resource allocation for multiple times.

Similar to our conclusion, prior machine learning based and heuristic-based systems also notice that they require to adjust the resource allocation for multiple times [17, 22, 27, 38, 41]. For instance, the bayesian optimization based system, ELIS [41], needs to search for 4-15 samplings to find the final resource configuration for each microservice. Reinforcement learning-based system, FIRM [38], has to perform multiple incremental model updates, if the microservice application has dynamics. Each adjustment interval is also at least several seconds, incurring more queued queries.

Even if the optimal resource allocations can be determined directly, the long monitoring interval and the blocking effect already result in the long query queues. We also evaluate the optimal resource decision case in Section 8.

3.4.3 Slow Query Draining

The queued queries during the monitoring and the resource adjustment period can result in the long QoS recovery time.

As an example, Figure 8(a) shows the 99%-ile latency of the benchmark *HR* with ELIS, when we change the dynamic scenario to *1.5X(2:1:2:1:4)*. As observed, although appropriate resources are allocated to each microservice for its "to-be-processed" load, the 99%-ile latency gradually drops from time 39.6 seconds to 47.9 seconds instead of backing to normal immediately. This is because the resource allocation does not consider the queued queries at each microservice.

We further try to allocate excessive resources for microservices, and explore the impact of excessive ratio on queued query draining. The excessive ratio is the excessive resource allocation ratio for microservices after the resource adjustment process. Figure 8(b) shows the draining time under different excessive ratios. We can observe that the larger the excessive ratio, the shorter the draining time.

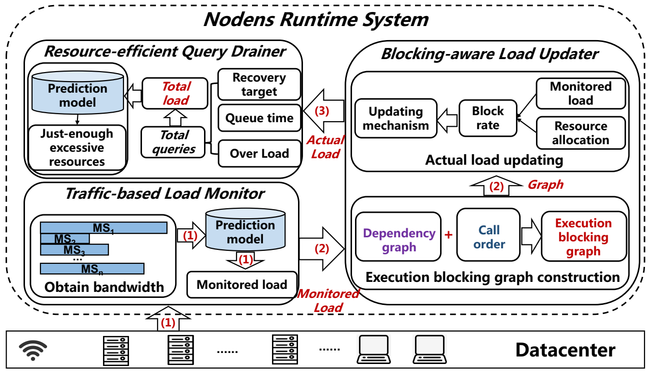


Figure 9: Design overview of Nodens.

It is possible to reduce the queued query draining time through excessive resource allocation, so that can reduce the overall QoS recovery time. We should carefully determine the amount of excessive resources to ensure high resource efficiency, while minimizing the QoS recovery time.

4 Nodens Methodology

We design **Nodens** to enable the fast QoS recovery of dynamic microservice applications based on the above analysis.

Figure 9 shows the design overview of Nodens. It comprises a *traffic-based load monitor*, a *blocking-aware load updater*, and a *resource-efficient query drainer*. The monitor predicts the monitored load of a microservice based on the periodically obtained network bandwidth usage of the microservice. The load updater calculates the actual "to-be-processed" load of each microservice, based on the monitored loads of the microservices, the resource allocation of the microservices, and the execution blocking graph of the application. The query drainer allocates "just-enough" excessive resources for each microservice, to quickly drain the queued queries generated during the above process.

We use the network bandwidth usage to predict the load of a microservice for the short monitoring time. Section 5 shows that the incoming network bandwidth of a microservice is closely related to its realtime load. The monitored bandwidth is stable with the 1 second interval. With the short and stable monitoring interval, we can find the load variation quickly and tune the resource allocation as early as possible, reducing the number of queued queries at a microservice when the load increases.

The most challenging part is obtaining the actual "to-be-processed" load of each microservice, due to the execution blocking effect. We define an *execution blocking graph* for a microservice application. It reflects the blocking relationship among microservices, and is determined by the microservice dependency graph and microservice call order. A load updating mechanism is also required to capture the complex realtime blocking actions due to the dynamics, based on the execution blocking graph (Section 6).

It is inevitable that some queries queue up at a microservice when its load increases, before the increase is noticed and the resource is adjusted. Without careful design, these queued queries result in serious QoS violations, or too much resource is allocated to handle the possible queued queries. The challenging part in the query drainer is to allocate just-enough excessive resources to ensure the QoS recovery time target while maintaining the resource efficiency (Section 7).

Specifically, Nodens manages the resource allocation of a microservice application in the following steps. 1) Nodens obtains the dependency graph and the possible call graphs of the application. Based on the obtained graphs, the *execution blocking graph* is built. 2) When serving the application, Nodens deploys a daemon process on each server to monitor the network usage of the microservices. 3) A server runs the load updater, and determines the resource allocation of each microservice with the query drainer. The load updater collects the bandwidth data of microservices on different servers, and calculates the actual "to-be-processed" load of every microservice. 4) The drainer updates the resource allocation accordingly, and sends back the allocation decision to each microservice. 5) The daemon process on each server then reallocates the resources based on the decision of the drainer.

Since the servers are in the same datacenter, we use gRPC [7] to collect the network usage of each microservice, and send back the allocation decision. The transfer latency is less than 5ms in our experiments. Nodens does not need to modify the source code of microservice applications, and can be implemented as a plug-in based on Kubernetes [14]. Moreover, Nodens does not focus on microservice deployment among distributed servers, and the initial deployment is determined by Kubernetes's random scheduling strategy.

Similar to prior works [29, 37, 41, 45], Nodens uses Linux cgroups [4] to adjust CPU resources, which can complete within 1ms. VPA [10] in Kubernetes also supports in-place pod vertical scaling with low overhead. After each allocation decision, if there are no resources available on some servers in the first place for vertical scale-up, Nodens utilizes the resource recycling idea [41] to deal with. Nodens will first recycle the resources from over-provisioned microservices on these servers, and then allocate them to microservices requiring scaling up. If some servers still lack sufficient resources for their deployed microservices after resource recycling, Nodens adopts current load balancing strategies [22, 33, 41] to migrate some microservices from busy servers to idle servers.

5 Traffic-based Load Monitor

5.1 The Speedup and Predictability

As discussed in Section 3.4.1, the latency monitoring can result in long resource adjustment time. Therefore, we use the upper network bandwidth usage for resource adjustment to reflect the load change of microservices. For microservice-6 of the *EB1* benchmark shown in Figure 4, its upper network bandwidth usage is the data communication amount per second from microservices 0, 2, and 3.

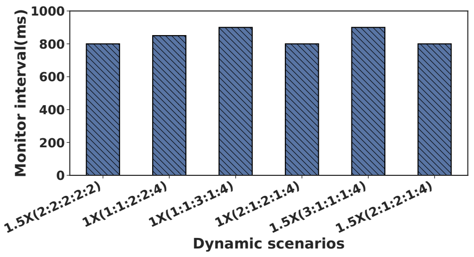


Figure 10: Network traffic monitoring intervals.

To explore the required interval of network traffic monitoring, we conduct 6 experiments on the *HR* benchmark, whose dynamic scenarios are the same as Section 3.3. For each dynamic scenario, we first run the benchmark and monitor all microservices for 10 seconds to obtain their stable network bandwidth usage (Mbits/s) as the baseline values. Then, we run the benchmark again and gradually increase the intervals (starting from 50ms) to find the minimum interval that may get stable monitoring data. We consider the monitoring data to be stable when the error between obtained microservices' bandwidth usage and corresponding baseline values are within 5%. Figure 10 shows that the obtained minimum monitoring intervals are less than 1000ms. Compared with the latency monitoring interval (3300ms) we test in Section 3.4.1, the required interval of monitoring network is 3X shorter.

In addition, we find the upper network bandwidth usage has a typical linear relationship with the load size (i.e., queries per second, QPS) for all microservices, and the relationship between load size and CPU core demand is the same. For a microservice application, we profile each call graph at 10 sets of loads (evenly from 0 to the peak supported load), and obtain the performance samples (i.e., the load, upper network bandwidth usage, and CPU core demand) of all microservices. The profiling can be done automatically and online. For long-running applications, call graphs can be known from history. Otherwise, we can trace the call graph and profile the new call graphs online. We use the profiled performance samples to train the linear models for microservices. We then use the performance samples at 10 other different sets of loads as the test dataset. Predicting the load size through the network bandwidth, and predicting the CPU core demand through the load size, the prediction accuracies are 97.0% and 97.9% on average for the 3 benchmarks, respectively. So, we can accurately predict the load size of each microservice through its upper network bandwidth, and further predict its CPU core demand. As prior works have shown that microservices are basically sensitive to CPU resources [31, 34, 45], Nodens primarily focuses on CPU core allocation. From our observations, the

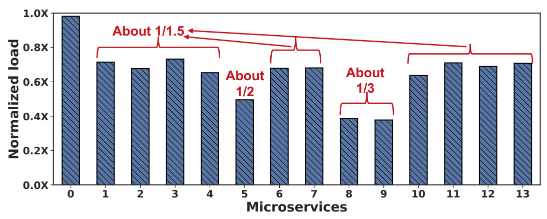


Figure 11: The monitored loads normalized to the actual loads of microservices.

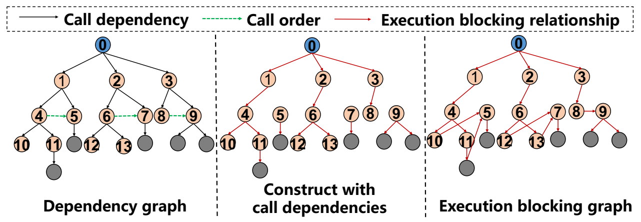


Figure 12: The dependency graph, and execution blocking graph of the *HR* benchmark.

memory usage of microservices is steady, and we pre-allocate enough memory capacity for microservices.

Compared with the ML-based methods [38,41], we can use these models to predict the CPU core demand from network bandwidth usage, whose overhead is usually less than 1ms.

5.2 Network Traffic Monitoring Methods

As the analysis of production microservice applications [31, 32], most of them have tree-like dependency graphs, while a few of them show graph-like structures. The in-degrees of some microservices in a graph-like structure are larger than 1.

For the tree-like dependency graph, we obtain the receive and transmit bytes of microservices' corresponding network interface during the monitoring interval by reading Linux file */proc/net/dev*, and then calculate the upper network bandwidth one by one based on the tree structure. The overhead of this method is less than 15ms, which includes reading the Linux file and calculations. For the graph-like dependency graph, we use Libpcap [12] to obtain network traffic between each pair of microservices directly, and then calculate the upper network bandwidth for each microservice. This method completes in 30ms. After obtaining the upper network bandwidth usage of microservices, the monitor uses linear regression models to predict the monitored loads of them as this module's output.

6 Blocking-aware Load Updater

6.1 Execution Blocking on Monitored Load

Since the effect of execution blocking under dynamics, the monitored loads may not equal to the actual "to-be-processed" loads of microservices. In this subsection, we explore the effect of execution blocking on monitored loads with the *HR* benchmark. We conduct an experiment from the initial state of *1X(2:2:2:2:2)* to the dynamic state of *1.5X(3:1:1:1:4)*. Figure 11 shows the normalized monitored loads to the actual loads of microservices. The left part of Figure 12 shows the dependency graph of *HR*.

As observed, since the load changes from 1X to 1.5X, we can find most microservices' monitored loads are about $\frac{1}{1.5}$ of their actual loads, as their superior microservices can only handle 1X load with current resource allocation. Microservices-8 and 9' monitored load is about $\frac{1}{3}$ of their actual loads, as the proportion of their located call graph also increases two times. Above observations prove that execution blocking effect can be caused by the **call dependencies among microservices**.

Moreover, microserivce-5's monitored load is about $\frac{1}{2}$ of its actual load, but not $\frac{1}{1.5}$. After looking into the dependency graph, we find since microservice-4 and microservice-5 are called by microservice-1 in a fixed order, the microservices in the subtree of microservice-4 can be the execution blocking microservices of microservice-5. In detail, since the load changes and the proportion of microservice-4's located call graphs also changes, microservice-4's subtree blocks $\frac{1}{2}$ of the actual load, which cannot be passed to microservice-5. Above observations prove that the execution blocking effect can be caused by the **call order among microservices**.

We give a simple example for Figure 12. Suppose queries pass through part of the microservices in following orders: 1) log in (microservice-3), 2) authentication (microservice-8), and 3) reservation (microservice-9). Suppose microservice-3's loads increase to 1500 queries per second but it only has just-enough resources to handle loads of 1000. At this point, the loads to its downstream microservices are blocked at 1000, which is 1/1.5 of 1500 (microservices-8 and 9's actual loads).

If we adjust resources only based on monitored loads, we need to adjust resources for microservices multiple times to deal with the execution blocking, which can greatly increase the QoS violation time. Therefore, a reasonable method is to combine the microservice dependency graph, monitored load, and resource allocation to update the actual loads of microservices in advance. During this process, we need to primarily consider the execution blocking effect caused by **call dependencies** and **call order** among microservices.

6.2 Execution Blocking Graph

Based on the observations in Section 6.1, we construct the **Execution Blocking Graph** for the microservice application.

Figure 13 shows the execution blocking graph construction based on the microservice dependency graph. The microservice dependency graph is obtained by using tracing tools (e.g., Jaeger [11]) after running the application online for one minute. Firstly, for the microservice in the dependency graph that has multiple in-degrees, we transform the subtree with it

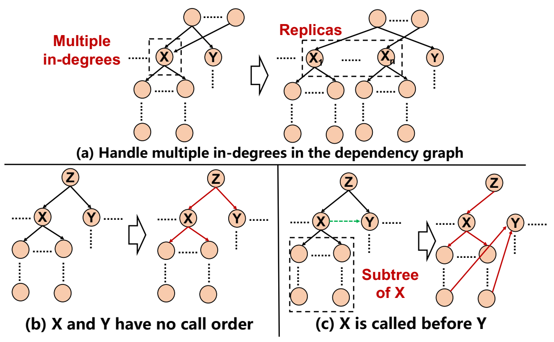

Multiple in-degrees
Replicas
(a) Handle multiple in-degrees in the dependency graph
(b) X and Y have no call order
Subtree of X
(c) X is called before Y

Figure 13: Execution blocking graph construction.

to multiple identical replicas, to maintain the tree structure of the dependency graph, as shown in Figure 13(a). The replicated overhead is low, because few microservices have multiple in-degrees in production microservice applications [31].

We then construct the execution blocking relationship for each sub structure. On the one hand, as shown in Figure 13(b), microservices X and Y have no fixed call order from their common superior microservice. In this case, the execution blocking relationship is equal to the call dependencies among microservices. On the other hand, as shown in Figure 13(c), X is called before Y by their common superior microservice. In this case, the execution blocking microservices of Y are the ones at the end of the execution blocking subtree with root X. The number of this kind of microservices can be one or more. We take an example for better explanations. Suppose X is the "authentication" microservice, Y is the "reservation" microservice, and Z is the "log in" microservice. As X is called before Y by Z, the resource insufficiency of X can block Y. By contrast, if X and Y are called by Z asynchronously, X cannot block Y.

Breaking down the microservice dependency graph into multiple sub structures, we iteratively build the execution blocking relationship from the root microservice. Then, we can obtain the execution blocking graph for the microservice application. Nodens does not need to replicate microservices with multiple in-degrees in the execution blocking graph.

We use the example of *HR* benchmark in Figure 12 to further explain the whole construction process. As microservices 1, 2, and 3 have no fixed call order, their execution blocking relationship is equal to their call dependencies. Some other microservices have similar construction, and the middle results are shown in the middle part of Figure 12. Moreover, microservice-5 is called after microservice-4 by their superior microservice-1. Therefore, its execution blocking microservices are the microservices at the end of the execution blocking subtree of microservice-4. Microservices 7 and 9 have similar construction process with considerations of the call order. The final execution blocking graph is shown in the right part of Figure 12.

The execution blocking graph differs from the execution graph in 2 ways. First, actually an execution graph is similar to a call graph, while the execution blocking graph is constructed once from the dependency graph and it captures all the possible blocking relationship. Second, we define the node and edge weights in the execution blocking graph for updating the actual loads for microservices (Section 6.3), while execution graphs do not have such information.

6.3 Actual Load Updating Mechanism

Based on the execution blocking graph, we then introduce the actual load updating mechanism.

We define the triple to record current state of each microservice i as $(MonitoredLoad_i, ActualLoad_i, HandleLoad_i)$. The $MonitoredLoad_i$ is obtained from the traffic-based load monitor. The $ActualLoad_i$ will be updated by the mechanism for each microservice, and is equal to the $MonitoredLoad_i$ at the beginning. The $HandleLoad_i$ represents the load that can be handled for each microservice. It is predicted by using the linear regression model from its corresponding microservice's resource allocation. For microservices i and j in the execution blocking graph, the edge weight EW_{ij} is defined as the load passing from i to j. The EW_{ij} is equal to the monitored load from i to j at the beginning, and will be updated during the updating mechanism.

Since microservices may block the load of their downstream microservices, we then define the blocking rate of the microservice j as:

$$rate_j = max(\frac{ActualLoad_j}{min(HandleLoad_j, MonitoredLoad_j)}, 1) \quad (1)$$

In this formula, the blocking rate is the $ActualLoad$ dividing the minimum of the $HandleLoad$ and $MonitoredLoad$, as the former may be larger than the latter since higher-level blocking. Moreover, as the loads of some microservices may decrease under dynamics, the blocking rate may be smaller than 1, and we set $rate_j = 1$ for these cases.

We mainly adopt the Breadth-First-Search (BFS) algorithm based on the execution blocking graph to calculate the blocking rate and update the actual loads of microservices layer by layer. Algorithm 1 shows the mechanism. We first initialize the load triple for microservices, the execution blocking graph, and a queue for the BFS process. We then put the root microservice into the queue, and then come into the major process of actual load updating. During this process, we first calculate the blocking rate of the head microservice j of the queue based on Eq.(1), whose $ActualLoad$ has been updated correctly, as shown in the lines of 6-8. Then, we start to handle the downstream microservices of j, as shown in the lines of 9-14. For each downstream microservice k, we first update the actual load from j to k with the blocking rate of j. If k's all entry edges are all updated, we then put it into the queue to follow the process of the BFS algorithm. The updating process is ended when the queue is empty, which represents the

Algorithm 1: Actual Load Updating Mechanism

```
 1: Initialize $(MonitoredLoad_i, ActualLoad_i, HandleLoad_i)$
 2: Initialize execution blocking graph $EBG$ with edge weights $EW_{ij}$
 3: Initialize a queue $q$ for the BFS process
 4: $q.put(EBG.root)$
 5: while $q \neq \emptyset$ do
 6:     $j = q.get()$
 7:     $ActualLoad_j = \sum_{i \to j} EW_{ij}$
 8:     $rate_j = max(\frac{ActualLoad_j}{min(HandleLoad_j, MonitoredLoad_j)}, 1)$
 9:     for each downstream microservice $k$ of $j$ do
10:         $EW_{jk} = EW_{jk} \times rate_j$
11:         if all entry edges of $k$ are updated then
12:             $q.put(k)$
13:         end if
14:     end for
15: end while
16: return $ActualLoads$ for all microserivces
```

actual loads of all microservices have been updated. Lastly, we return the actual loads of microservices.

7 Resource-efficient Query Drainer

Although the resource adjustment time of Nodens is greatly decreased compared to the latency based resource adjustment methods, the queued query draining is non-negligible, as just-enough resources can lead to the queued queries being unable to be drained for a long time. As discussed in Section 3.4.3, the larger amount of excessive resources can accelerate the draining process, but obviously sacrifice the resource efficiency. We define the QoS recovery time as the time needed to reduce the 99%-ile latency to be below a fixed latency target (e.g., 100ms) after microservice dynamics happen. Same to prior work [21, 38, 41, 45], the QoS is often defined to be latency. Nodens can also support other QoS definitions (e.g., throughput) through simple adaption. We set the QoS recovery time target for the microservice application, e.g., the QoS recovery time is within 3 seconds after microservice dynamics happen. Therefore, the excessive resource allocation of the microservice application can be described as *minimizing the excessive resource allocation on the premise of guaranteeing the recovery time target*.

To allocate just-enough excessive resources for each microservice, we try to drain the queued queries for each microservice exactly within the recovery time target. Therefore, our goal is to calculate the total queries to be processed for each microservice during the residual recovery time, and then allocate just-enough excessive resources correspondingly.

We first calculate the overload of each microservice i during the resource adjustment process as:

$$OverLoad_i = ActualLoad_i - min(MonitoredLoad_i, HandleLoad_i) \tag{2}$$

where the $ActualLoad_i$, $MonitoredLoad_i$, and $HandleLoad_i$ have the same definition to Section 6. For some microservices, their actual loads may be less than or equal to their monitored and handle loads under microservice dynamics. For these cases, we set $OverLoad_i = 0$ for them.

After calculating the $OverLoad_i$, we can further calculate the total queries to be processed for each microservice during the residual recovery time as:

$$TotQuery_i = OverLoad_i \times T_i + ActualLoad_i \times (QT - T_i) \tag{3}$$

where T_i is the resource adjustment time which causes query queuing, and QT is the recovery time target. In this formula, the first item represents the total amount of queued queries, while the second represents the total amount of normal queries that need to be processed during the residual recovery time.

At last, we can calculate the total load (i.e., queries per second) that needs to be handled during the residual QoS recovery time for each microservice i as:

$$TotLoad_i = \frac{TotQuery_i}{(QT - T_i)} \tag{4}$$

Obtaining the $TotLoad$, we can use the linear model mentioned in Section 5.1 to predict the total CPU core demand of each microservice, and then tune the allocated resources accordingly. The total CPU core demand includes the just-enough resources under the corresponding dynamic scenario and excessive resources for queued query draining. After QoS is recovered, the excessive resources will be recycled to maintain high resource efficiency.

8 Evaluation of Nodens

In this section, we first evaluate the performance of Nodens in recovering the QoS while achieving resource efficiency. Then, we show the effectiveness of the blocking-aware load updater, and the resource-efficient query drainer.

8.1 Evaluation Setup

Table 1 already shows the configurations of the experimental platform. We evaluate all the three benchmarks *HR*, *EB1*, and *EB2* on the eight-node cluster in the experiments. For each benchmark, we evaluate Nodens with six dynamic scenarios, including load dynamic, call graph dynamic, and the mix of the two types of dynamics.

We compare Nodens with two state-of-the-art microservice management systems FIRM [38] and ELIS [41]. FIRM monitors the latencies of microservices periodically, identifies the critical path and critical microservices, and increases their resources to the optimal resource allocations using reinforcement learning. ELIS first recycles the over-provisioned resources of non-critical microservices before reallocating the resources. It uses bayesian optimization to reallocate resources. For FIRM and ELIS, the latency monitoring periods

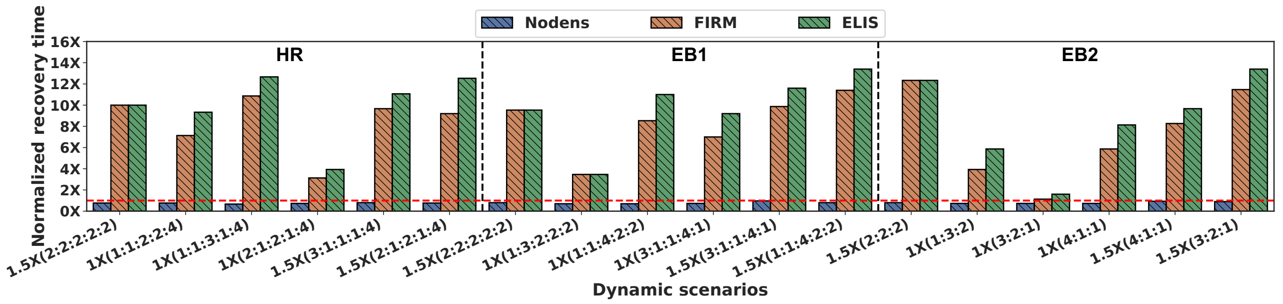


Figure 14: The normalized QoS recovery time relative to the recovery time target of benchmarks with Nodens, FIRM, and ELIS.

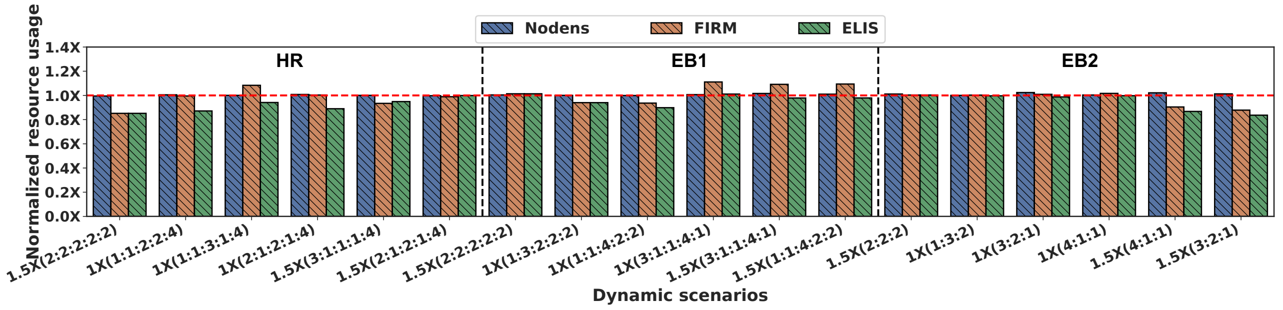


Figure 15: The normalized resource usage relative to the just-enough resources of benchmarks with Nodens, FIRM, and ELIS.

are set to be the minimum time with which the latency is stabilized (i.e., the subsequent latency increase is less than 5%). In Nodens, we use 1 second to be the network traffic monitoring interval as analyzed in Figure 10.

In our experiments, we already optimize FIRM and ELIS through offline profiling. While ML-based resource adjustment requires multiple iterations to find the optimal resource configuration, we optimize them to be able to directly allocate the optimal resources for microservices. Moreover, we also give each microservice excessive resources according to the recommendation of Nodens's query drainer. The native FIRM and ELIS perform worse than the ones we used here.

In the following experiments, we use 3 seconds to be the recovery time target. In other words, Nodens will adjust the resource allocation, in order to make sure that the QoS violation is eliminated in 3 seconds.

8.2 QoS Recovery and Resource Efficiency

In all test cases, microservices are initially allocated the just-enough resources when the load is 1X and the percentages of queries that go to different call graphs are identical. Then, we change the loads of the entire benchmark and the percentages of queries go to different call graphs, and evaluate the performance of Nodens in recovering the QoS before the given QoS recovery target.

Figure 14 shows the QoS recovery time of all the $3 \times 6 = 18$ test cases with FIRM, ELIS, and Nodens, respectively. The time is normalized to the recovery time target (3 seconds). In the figure, "1.5X(2:2:2:2:2)" represents the case that the application's load increases to 1.5X, and the percentages of queries to the five call graphs are identical. As observed from the figure, Nodens successfully eliminates the QoS violation in the given recovery time target. By contrast, the QoS recovery time with FIRM and ELIS is 7.9X and 9.4X of the recovery time target, and 10.2X and 12.1X of Nodens's.

Nodens has shorter QoS recovery time because it has shorter but stable load monitoring interval, and calculates the actual "to-be-processed" load of each microservice. It is able to reduce the queued queries during the monitoring interval, and allocate enough resources for each microservice beforehand. We can also find that the QoS recovery time is longer with ELIS than with FIRM. This is because ELIS first recycles the over-provisioned resources, which can spend some extra time. Moreover, the QoS recovery time is also short with ELIS and FIRM in some cases (e.g., the scenario *1X(3:2:1) with EB2*). It happens when there are only a few microservices' resources are insufficient.

Figure 15 shows the corresponding total resource usage (*cores* $\times$ *hours*) of the test cases during the QoS recovery process. The resource usage is normalized to the case that all the microservices have "just-enough" resources for the new load since the dynamic happens. We use the longest QoS recovery time (i.e., ELIS's) to calculate the total resource usage for the fair comparison. As observed, Nodens uses 1.5% and 6.1% more resources on average than FIRM and ELIS, respectively.

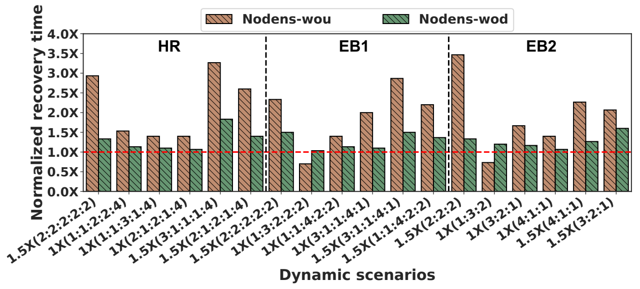


Figure 16: The QoS recovery time with Nodens-wou and Nodens-wod normalized to the recovery time target.

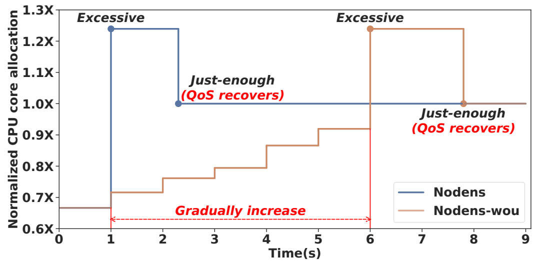


Figure 17: The CPU core allocation timeline of Nodens and Nodens-wou in an example dynamic scenario.

FIRM also uses more resources than ELIS. This is because FIRM only increases the resources of critical microservices without recycling the over-provisioned resource.

Therefore, Nodens is resource efficient while realizing the fast QoS recovery.

8.3 Effectiveness of the Load Updater

In this experiment, we show the performance of *Nodens-wou*, a variant of Nodens that disables the blocking-aware load updater. With Nodens-wou, the query drainer still allocates excessive resources for the microservices.

The orange bars of Figure 16 show the QoS recovery time of all the test cases with Nodens-wou normalized to the QoS recovery time target. As observed, Nodens-wou recovers the QoS before the recovery time target in only two cases. Compared with Nodens, Nodens-wou requires 2.6X time on average to recover the QoS.

As an example, Figure 17 shows the normalized resource allocation timeline of Nodens and Nodens-wou in the test case *1.5X(2:1:2:1:4)* of the *HR* benchmark. Other test cases show similar conclusions. As shown in the figure, Nodens allocates excessive resources to the microservices at an early time, and returns to the "just-enough" resource allocation once the QoS violation is eliminated. On the contrary, Nodens-wou gradually increases the resource allocated to the microservices after each monitoring interval. This is because the execution blocking effect makes Nodens-wou cannot obtain the actual "to-be-processed" loads of the microservices. For a

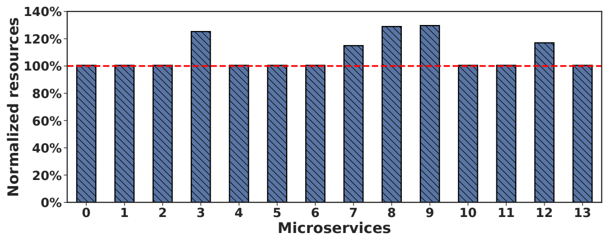


Figure 18: The resource allocation of microservices normalized to the just-enough resources for an example experiment.

microservice, its load pressure is released layer by layer from its superior microservices.

The blocking-aware load updater is necessary for Nodens. It avoids the execution blocking effect by updating the actual "to-be-processed" loads of microservices in advance.

8.4 Effectiveness of the Query Drainer

In this experiment, we show the performance of *Nodens-wod*, a variant of Nodens that disables the query drainer. The blocking-aware load updater still works in Nodens-wod.

The green bars of Figure 16 show the QoS recovery time of all the test cases with Nodens-wod normalized to the QoS recovery time target. As observed, Nodens-wod fails to recover the QoS before the recovery time target in all the cases. Compared with Nodens, Nodens-wod requires 1.6X time on average to recover the QoS. This is because the queued queries generated during the resource adjustment cannot be drained up quickly without the excessive resources allocated by the query drainer. Moreover, we can find that Nodens-wod performs better than Nodens-wou in most cases. This is because Nodens-wod can eliminate the execution blocking effect which is the most important influence factor that causes long-time QoS violations.

As an example, Figure 18 shows the actual resources allocated of microservices normalized to the just-enough resources in the test case *1X(1:1:3:1:4)* of the *HR* benchmark with Nodens. As observed, Nodens allocates excessive resources for 5 of the 14 microservices in the test case. Specifically, microservices 3, 8, and 9 belong to the same call graph and are affected by the same degree of execution blocking, so they have almost the same ratio of excessive resources. Microservices 7 and 12 belong to another call graph, and their resource shortage is smaller than the previous 3 microservices, so Nodens's drainer allocates them less excessive resources.

8.5 The Impacts of Dynamics

Since there are load dynamics, call graph dynamics, and load+call graph dynamics when serving a microservice application, we evaluate their impacts on the query drainer.

Figure 19 shows the ratio of total excessive resource allocation for all dynamic scenarios of the 3 benchmarks with

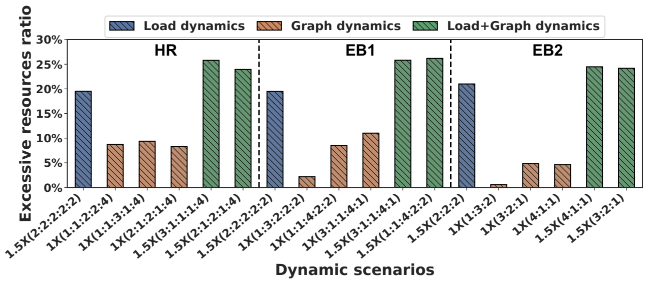


Figure 19: The ratio of total excessive resource allocation for all dynamic scenarios of the three benchmarks.

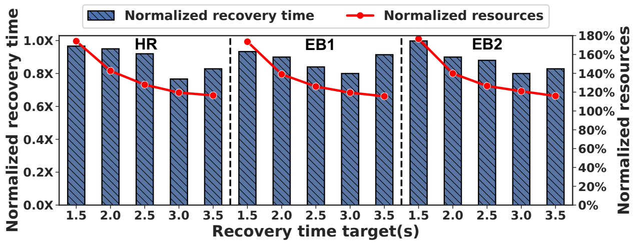


Figure 20: The QoS recovery time and resource allocation under different recovery time targets.

Nodens. As observed, the excessive resource ratio is smallest with only call graph dynamics, larger with only load dynamics, and the largest with both load and call graph dynamics.

This is because dynamic call graph scenarios only cause a few microservices' resources to be insufficient and dynamic load scenarios cause more, while the simultaneous dynamic load and call graph scenarios cause the most. As the resource shortage of more microservices can cause more queries queued, Nodens' drainer will allocate more excessive resources for these scenarios.

8.6 Handling Different Recovery Time Targets

In this experiment, we show Nodens's performance in handling different recovery time targets. We use one dynamic case of each benchmark to conduct the experiment, i.e., 1.5X load with identical call graph percentages.

Figure 20 shows the QoS recovery time and the actual resource allocation for all the cases. The QoS recovery time and the actual resource allocation are normalized to the recovery time target and just-enough resources in each case, respectively. As observed, Nodens successfully eliminates the QoS violation in different given recovery time targets for all the cases. Moreover, Nodens allocates more/fewer resources for the case with the smaller/larger recovery time target, which proves Nodens's resource efficiency.

Therefore, Nodens can ensure different QoS recovery time targets, while maintaining resource efficiency.

8.7 Overhead of Nodens

Offline Overhead. To train the prediction models for CPU core allocation, Nodens needs to profile the bandwidth and performance data for different microservices at different loads in advance. The offline profile time is about 25 minutes for each benchmark. The offline training time of the linear regression models for each benchmark is less than 150 ms.

Online Overhead. After deploying Nodens online, the execution time of the load monitor to get network traffic is less than 30ms. Moreover, the execution time for the load updater and query drainer are both less than 5ms. The prediction time and CPU core allocation time are all less than 1ms. The data transfer latency between servers is less than 5ms. Therefore, the total online overhead is about 50ms. The overhead is acceptable as it is far less than the monitoring interval in our experiments (i.e., 1 second).

We also evaluate Nodens's overhead using a simulated large-scale application with 200 microservices and 10 call graphs based on production-level microservice traces [31]. The online overhead is 126.6ms (59.4ms, 31.6ms, and 35.6ms for the network monitor, load updater, and query drainer, respectively), the offline profiling overhead is 50 minutes, and models can be trained in 2s.

9 Conclusion

In this paper, we propose Nodens to enable fast QoS recovery of dynamic microservice applications, while maintaining the efficiency of resource usage. Nodens's traffic-based load monitor predicts the monitored loads for microservices based on their network bandwidth usage. Nodens's blocking-aware load updater calculates the actual "to-be-processed" loads of microservices based on the execution blocking graph. It can eliminate the execution blocking effect, so that can reduce the total resource adjustment time. The query drainer allocates excessive resources for microservices to drain the queued queries, ensuring the QoS recovery time target. We have implemented Nodens and the experimental results show that, compared to the state-of-the-art microservice management systems, Nodens reduces the QoS recovery time by 12.1X.

Acknowledgment

We sincerely thank Mohammad Shahrad and our anonymous reviewers, for their helpful comments and suggestions. This work is partially sponsored by the National Natural Science Foundation of China (62232011, 62022057, 61832006), and Shanghai international science and technology collaboration project (21510713600). Quan Chen and Minyi Guo are the corresponding authors.

References

[1] Adopting microservices at netflix: Lessons for architectural design. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.

[2] Alibaba microservice cluster trace v2021. https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2021.

[3] Case studies of Sina Weibo. https://www.alibabacloud.com/help/en/function-compute/latest/sina-weibo.

[4] cgroups(7) — linux manual page. https://man7.org/linux/man-pages/man7/cgroups.7.html.

[5] Design Twitter — Microservices Architecture of Twitter Service. https://thinksoftware.medium.com/design-twitter-microservices-architecture-of-twitter-service-996ddd68e1ca.

[6] Golearn. https://github.com/sjwhitworth/golearn.

[7] gRPC. https://grpc.io/.

[8] Huawei Cloud Geo-distributed VM Traces. https://github.com/shijiuchen/HuaweiCloud-GeoVMTraces.

[9] Implementing microservices on AWS. https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/microservices-on-aws.html.

[10] In-place update of pod resources. https://github.com/kubernetes/enhancements/issues/1287.

[11] Jaeger: open source, end-to-end distributed tracing. https://www.jaegertracing.io/.

[12] Libpcap 1.x.y by the tcpdump group. https://github.com/the-tcpdump-group/libpcap.

[13] Porter Stemmer for Go. https://github.com/a2800276/porter.

[14] Production-grade container orchestration. kubernetes.io.

[15] QuickSort - Go Packages. https://pkg.go.dev/github.com/TannerGabriel/learning-go/algorithms/sorting/QuickSort.

[16] Weighted PageRank implementation in Go. https://github.com/alixaxel/pagerank.

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

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

[19] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 153–167, 2017.

[20] Weihao Cui, Han Zhao, Quan Chen, Hao Wei, Zirui Li, Deze Zeng, Chao Li, and Minyi Guo. DVABatch: Diversity-aware Multi-Entry Multi-Exit batching for efficient processing of DNN services on GPUs. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 183–198, 2022.

[21] Kaihua Fu, Jiuchen Shi, Quan Chen, Ningxin Zheng, Wei Zhang, Deze Zeng, and Minyi Guo. QoS-aware irregular collaborative inference for improving throughput of dnn services. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 993–1006. IEEE Computer Society, 2022.

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

[23] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 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*, pages 135–151, 2021.

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

[25] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer:

Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*, pages 19–33, 2019.

[26] Alim Ul Gias, Giuliano Casale, and Murray Woodside. ATOM: Model-driven autoscaling for microservices. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1994–2004. IEEE, 2019.

[27] Md Rajib Hossen, Mohammad A Islam, and Kishwar Ahmed. Practical efficient microservice autoscaling with QoS assurance. In *In Proceedings of HPDC*, 2022.

[28] Zijun Li, Quan Chen, Shuai Xue, Tao Ma, Yong Yang, Zhuo Song, and Minyi Guo. Amoeba: Qos-awareness and reduced resource usage of microservices with serverless computing. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 399–408. IEEE, 2020.

[29] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. RunD: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 53–68, 2022.

[30] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, et al. Help rather than recycle: Alleviating cold startup in serverless computing through Inter-Function container sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 69–84, 2022.

[31] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 412–426, 2021.

[32] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, and Chengzhong Xu. An in-depth study of microservice call graph and runtime performance. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):3901–3914, 2022.

[33] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, Guodong Yang, and Chengzhong Xu. 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*, pages 62–77, 2022.

[34] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu. The power of prediction: microservice auto scaling via workload learning. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 355–369, 2022.

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

[36] Amirhossein Mirhosseini, Brendan L West, Geoffrey W Blake, and Thomas F Wenisch. Q-zilla: A scheduling framework and core microarchitecture for tail-tolerant microservices. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 207–219. IEEE, 2020.

[37] Pu Pang, Quan Chen, Deze Zeng, and Minyi Guo. Adaptive preference-aware co-location for improving resource utilization of power constrained datacenters. *IEEE Transactions on Parallel and Distributed Systems*, 32(2):441–456, 2020.

[38] Haoran Qiu, Subho S Banerjee, Saurabh Jha, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 805–825, 2020.

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

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

[41] Jiuchen Shi, Jiawen Wang, Kaihua Fu, Quan Chen, Deze Zeng, and Minyi Guo. QoS-awareness of microservices with excessive loads via inter-datacenter scheduling. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 324–334. IEEE, 2022.

[42] Akshitha Sriraman and Thomas F Wenisch. μ suite: a benchmark suite for microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12. IEEE, 2018.

[43] Tetsuro Yamamoto. Historical developments in convergence analysis for Newton's and Newton-like methods. *Journal of Computational and Applied Mathematics*, 124(1-2):1–23, 2000.

[44] Wei Zhang, Quan Chen, Kaihua Fu, Ningxin Zheng, Zhiyi Huang, Jingwen Leng, and Minyi Guo. 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*, pages 570–582, 2022.

[45] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G Edward Suh, and Christina Delimitrou. 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*, pages 167–181, 2021.

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