2024 IEEE 6th International Conference on Cognitive Machine Intelligence (CogMI) | 979-8-3503-8672-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/CogMI62246.2024.00024

A Study of Response Time Instability of Microservices at High Resource Utilization in the Cloud

Qingyang Wang
Louisiana State University
Baton Rouge, USA
qwang26@lsu.edu

Xuhang Gu
Louisiana State University
Baton Rouge, USA
xgu5@lsu.edu

Calton Pu
Georgia Institute of Technology
Atlanta, USA
calton@cc.gatech.edu

***Abstract*—Maintaining consistently low response times is a critical performance requirement for mission-critical, web-facing applications (e.g., e-commerce) that typically use microservices architectures. Through extensive benchmarking of a microservices application in a cloud environment, we demonstrate that its response time stability is fragile, with significant variations (ranging from milliseconds to seconds) when the CPU utilization of component servers reaches moderate to high levels (e.g., 60%). Our detailed timeline analysis identifies a leading cause of response time instability at these utilization levels: a ripple effect caused by the long chain of dependencies inherent in microservices applications. In such an architecture, a millibottleneck (a bottleneck lasting sub-seconds) can trigger a queuing effect from a downstream server that propagates to upstream servers, resulting in dropped requests and TCP retransmissions lasting several seconds at the weakest link in the chain. We demonstrate two common factors that contribute to millibottlenecks in microservices runtime environments: interference from collocated containers and bursty workloads, both of which are prevalent in real-world cloud environments.**

***Index Terms*—response time instability, scalability, microservices, millibottlenecks.**

I. Introduction

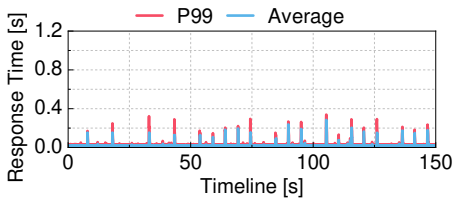
Fast response time is important for mission-critical web-facing applications (e.g., e-commerce) due to its significant business impact. For example, Amazon has reported that a 100-millisecond increase in page load time can lead to a 1% drop in sales [18]. Google revealed that a 500-millisecond delay in displaying search results could result in a 20% decline in revenue [17]. Augmented-reality devices such as Microsoft HoloLens also need system backend responsiveness in order to guarantee smooth and natural interaction [9]. Thus, always maintain fast response time and avoid any performance instability means large money incentive for both business owner and cloud computing infrastructure vendors.

At the same time, web-facing application architecture is transitioning from traditional monolithic n-tier designs to lightweight, loosely coupled microservices [10], [23]. This shift is driven by the benefits of microservices, such as better scalability, easier cross-team development, and simpler deployment. However, breaking down a monolithic system into hundreds or thousands of smaller microservices creates complex internal dependencies among component microservices, making it more challenging to predict and manage performance [25].

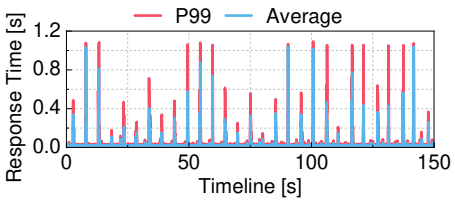
In this paper, we show an empirical study of response time instability of microservices, through extensive measurements of a social network application benchmarks [11] in a cloud environment. Our study shows that the chain of inter-service dependencies could be a major cause of long response time requests (on the order of seconds). The chain of dependencies commonly exists in a microservice application where component services adopt the Remote Procedure Calls (RPCs) for inter-server/service communication [29], through the use of RPC frameworks such as Google's gRPC [3] or Facebook/Apache's Thrift [2]. We have found that RPC brings a long call dependency chain as requests traversing from one component microservice to another. We found that a small queuing effects from a downstream microservice caused by a millibottleneck (a bottleneck lasting sub-seconds) can be propagated to its upstream microservices along the long call chain of dependencies. Such a cross-service queue propagation will lead to queue overflow and dropped requests in the weakest link along the long call chain of dependencies, causing TCP retransmissions and requests with long response time up to seconds.

We illustrate two common factors that can cause millibottlenecks in a runtime microservices application: interference from collocated containers and bursty workloads, both of which are prevalent in real-world cloud environments. Specifically, we observed significant variations in application response times due to these factors when the resource utilization of certain component microservices reaches moderate to high levels. For example, a millibottleneck caused by interference from collocated containers in a database service can propagate queues and lead to dropped requests in an app service located two hops away from the original microservice where the millibottleneck occurs. To exacerbate the issue, we observed that large numbers of dropped requests often synchronize, causing multiple waves of queue overflows over a span of ten seconds. This occurs because the dropped requests follow the same timeout mechanism, causing the retransmitted

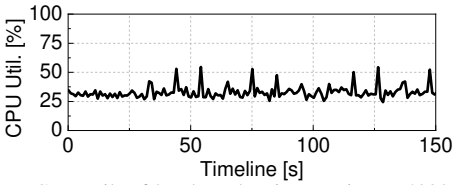
DOI 10.1109/CogMI62246.2024.00024



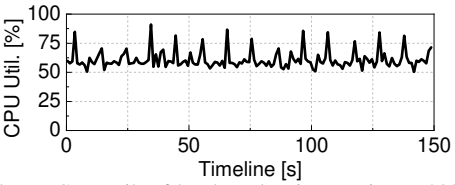
(a) Moderate response time fluctuations at 4000 clients



(b) Large response time fluctuations at 8000 clients



(c) Low CPU util. of bottleneck microservice at 4000 clients



(d) Moderate CPU util. of bottleneck microservice at 8000 clients

Fig. 1: Response time instability was observed when the CPU utilization of the busiest microservice in the system reached moderate to high levels. (a) and (c) display the results with 4,000 clients, while (b) and (d) show the results with 8,000 clients.

requests to arrive at nearly the same time, triggering another wave of burst requests, leading to further queue overflow and dropped requests at the weakest link. In our second example of millibottlenecks caused by a large burst of traffic, a single millibottleneck can cause three waves of transient queue overflows and request drops, resulting in significant response time instability.

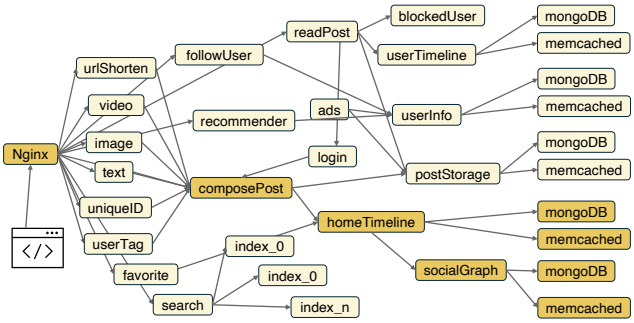
Overall, our study provides an explanation for the commonly observed low average utilization of cloud data centers (20% to 50% [26], [30], [31], [36]). Maintaining low utilization is often an effective way to meet response time stability objectives specified in SLAs, especially when more precise knowledge is lacking. Given the numerous factors at different system levels that can cause millibottlenecks, maintaining response time stability of microservices applications becomes increasingly challenging at moderate high utilization levels.

II. Background and Motivation

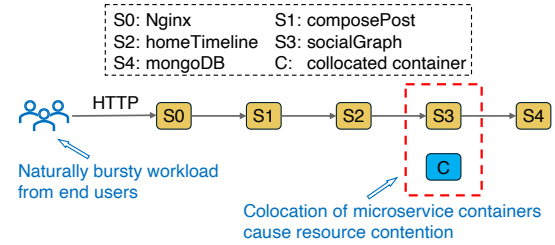
A. Background

Long Call Chain in Microservices. Microservice applications often consist of numerous microservices that communicate via RPC calls. These calls create extensive dependency chains as a single request traverses multiple microservices before returning a response. For example, a published trace from Alibaba [23] reveals that over 10% of its microservice call chains involve more than 40 unique microservices, creating complex runtime dependencies. The trace also indicates that over 30% of the call chains have a depth greater than 4, 20% exceed a depth of 5, and in extreme cases, around 10% have depths greater than 10.

These long call chains introduce complex runtime dependencies that can lead to performance instability. As we demonstrate in this paper, a queuing issue in a downstream microservice can propagate to all its upstream microservices in the call chain, potentially causing queue overflows and dropped requests at the weakest link, resulting in TCP retransmissions and very long response times.



(a) SocialNetwork benchmark microservices topology. Every component microservice runs in a dedicated container.



(b) A representative long call chain of dependencies in SocialNetwork. A microservice container may collocate with containers from other applications in the same VM, causing performance interference.

Fig. 2: SocialNetwork benchmark execution dependency graph and experimental setup

Queue Overflow and TCP Retransmissions. When a microservice experiences resource saturation, its application-level queue (e.g., message queue or thread pool) fills up. Once this queue is full, the microservice stops accepting new requests, causing additional requests to be queued in its TCP backlog. If the TCP backlog also fills up, any further requests will be dropped, triggering TCP retransmissions.

The TCP retransmission timeout (RTO) is typically cal-

culated based on the round-trip time (RTT) between the sender and receiver. The minimum RTO is set to 1 second and doubles with each subsequent retransmission [6]. For instance, if the RTT is 100ms, the RTO will be rounded up to 1 second. If a request is dropped, the sender waits 1 second before retransmitting. If the retransmitted request is also dropped, the sender waits 2 seconds before trying again. This exponential backoff mechanism helps prevent network congestion and reduces excessive retransmissions, but it can also cause significant delays in receiving a response, especially if a request is unfortunately dropped multiple times.

Benchmark Application. We adopt the Social Network benchmark from DeathStarBench [11] as our benchmark application. The Social Network benchmark is a microservices-based application that simulates a social media platform. The benchmark consists of more than 30 microservices (as shown in Fig. 2a), each of which performs a specific function, such as user authentication, posting messages, and generating social graphs. The microservices communicate with each other over RPC calls. We deploy the benchmark on a Kubernetes cluster, where each microservice is hosted in a container. We adopt the classic RUBBoS workload generator [5] to simulate a configurable number of normal users navigating the website.

To induce internal colocation interference, we explicitly colocate a container running CPU-intensive computation tasks with containers hosting the benchmark microservices application. These tasks are CPU-bound and consume 100% of the available CPU resources. To induce external bursty workloads, we use the Httperf [4] benchmarking tool to generate a burst of traffic by sending a sudden influx of requests to our benchmark microservices application, simulating a spike in user activity.

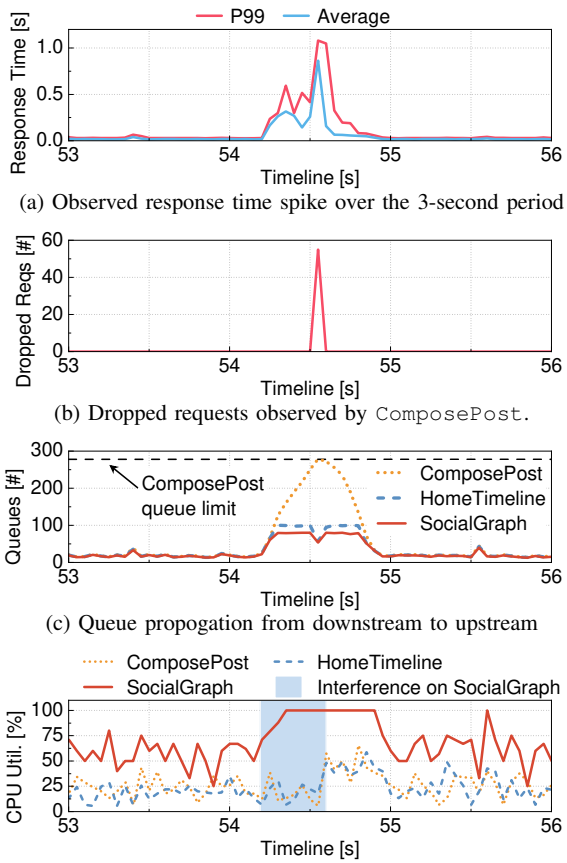
B. Motivation experimental results

To highlight the significance of the issue, we create internal colocation interference to a microservice `SocialGraph` as shown in Fig. 2b. Fig. 1a and 1b show 99th percentile and average end-to-end response time of the system at a moderate and a high workload, respectively. Fig. 1a shows that the system already exhibits some minor response time fluctuation at workload 4000 normal users. Fig. 1b shows that the system encounters significant response time fluctuations, ranging from milliseconds to over a second, despite average CPU utilization being around 55%—well below saturation. Such large response time fluctuations can lead to poor user experience and potentially large business revenue loss [17], [18]. In the following sections, we will explain two factors that cause the large response time fluctuations: internal colocation interference and external bursty workloads.

III. Millibottlenecks Caused by Internal Colocation Interference

Internal colocation interference occurs when multiple microservices share the same physical host, leading to resource contention and performance degradation. Microservices are typically implemented with containers, which are lightweight and portable units of software that package an application and its dependencies [16]. Containers are often deployed on shared or multi-tenant compute resources, such as virtual machines (VMs) or physical servers. When multiple containers share the same physical host, they must compete for resources, such as CPU, memory, and network bandwidth. This competition can lead to resource contention and performance degradation, especially when one or more containers consume a large amount of resources [22], [25], [37].

To demonstrate the impact of internal colocation interference on the response time of a microservices application, we explicitly colocate a container running heavy computation tasks with a container running the `SocialGraph` microservice, as shown in Fig. 2b. The heavy computation tasks are CPU-bound and consume 100% of the CPU resources.



(a) Observed response time spike over the 3-second period

(b) Dropped requests observed by `ComposePost`.

(c) Queue propogation from downstream to upstream

(d) CPU utilization of the microservices along the call chain. A millibottleneck was observed in the downstream service `SocialGraph`.

Fig. 3: **A fine-grained analysis of the experiments shown in Fig. 1b and 1d over a 3-second time span, showing the impact of a millibottleneck on the long call chain (see Fig. 2b).** A millibottleneck caused by interference from a collocated container on the downstream microservice `SocialGraph` (d) triggers queue propagation to upstream services (c), leading to request drops in the upstream microservice `ComposePost` (b). This results in prolonged response times, as seen in (a), due to TCP retransmissions.

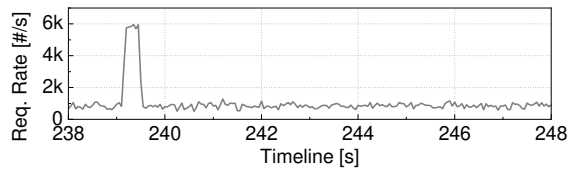
Fig. 3 shows a fine-grained (with a 50ms time window) analysis of the experiments shown in Fig. 1b and 1d in a 3-second time span. Fig. 3d shows the CPU utilization of the microservices along the call chain. The millibottleneck occurs on SocialGraph caused by interference from collocated container. The local queue of SocialGraph soon fills up during the millibottleneck period, pushing requests to queue in the upstream microservices (HomeTimeline and ComposePost), as shown in Fig. 3c. Once the queued requests reach limit of ComposePost (about 280, including both application-level queue and TCP backlog), the requests will be dropped, resulting in TCP retransmissions. Fig. 3b shows the number of dropped requests by ComposePost during the same 10-second time period. The periods of queue overflow by ComposePost in Fig. 3c and requests drop in Fig. 3b match well with each other. Since initial retransmission timeout is 1 second, the dropped requests will be retransmitted after 1 second. As a result, the total response time of those dropped requests will be more than 1 seconds. Fig. 3a shows the response time of the system during the interference period.

In this representitive experiments, it is challenging to set a proper queue size of the microservice ComposePost since it has more than 7 ingress upstream microservices (see Fig. 2a). The total incoming requests can be much greater than the number of concurrent requests that ComposePost can handle, especially when some queue slots are occupied by downstream microservices during millibottlenecks.

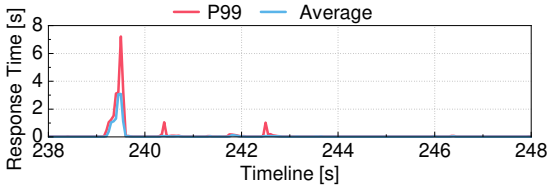
IV. Millibottlenecks Caused by External Bursty Workload

External bursty workloads occur when a microservice receives a sudden increase in requests, causing the system to become temporarily overloaded and response times to spike. Bursty workloads are common in web-facing applications, such as social media, e-commerce sites, and search engines, where user traffic can vary significantly throughout the day [8], [28]. Bursty workloads can lead to unpredictable response times and degraded performance, especially when the system is already under heavy load [13], [20].

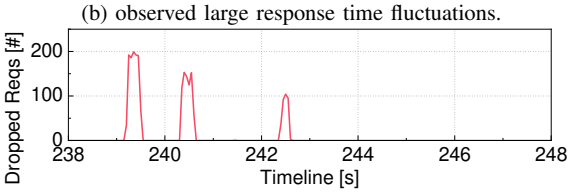
Fig.4 shows a representative 10-second snapshot with each metric monitored at a high resolution (with a 50ms time window). This figure illustrates how a bursty workload can cause large response time fluctuation. Fig. 4a shows a burst in the workload occurs in around 239s. The burst cause transient CPU saturation of the bottleneck microservice (SocialGraph) in Fig. 4e. The transient CPU saturation creates a millibottleneck (less than 1 second) and causes requests to queue in SocialGraph. The local queue soon fills up during the millibottleneck period, pushing requests to queue in the upstream microservices (HomeTimeline and ComposePost), as shown in Fig. 4d. Similar to the internal colocation interference in Section III, once the queued requests in ComposePost reach limit (including both application-level queue and TCP backlog), the new coming requests will be dropped, resulting in TCP retransmissions. Since the large amount dropped requests follow the same timeout



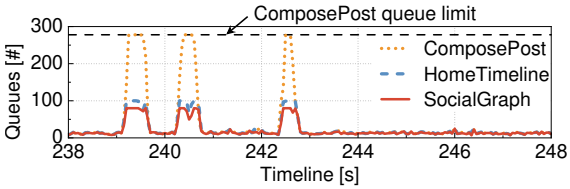
(a) A single burst of traffic end users



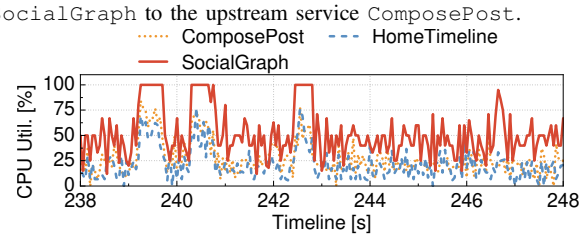
(b) observed large response time fluctuations.



(c) Multiple waves of dropped requests by ComposePost.



(d) Queue propogation from the downstream bottleneck service SocialGraph to the upstream service ComposePost.



(e) CPU utilization of the microservices along the call chain. Multiple millibottlenecks were observed in SocialGraph. The first one (around 239s) is caused by the initial traffic burst; the second and third are caused by the following synchronized retransmitted requests.

Fig. 4: An illustration showing how a single burst of traffic triggers multiple waves of dropped requests and TCP retransmissions over a ten-second span, resulting in significant response time fluctuations.

mechanism, causing the retransmitted requests to arrive at nearly the same time, triggering another wave of burst requests and queue overflow by ComposePost. This cycle repeats multiple times, causing multiple waves of queue overflow and dropped requests as shown in Fig. 4c. The high response time spikes of the system is shown in Fig. 4b. In this example, the requests that experience the worst case of response time are retransmitted 3 times, leading to a response time of more than 7 seconds.

V. Related Work

Queue Management in Microservices. Optimizing queue

size can effectively reduce queue overflows and improve the performance of internet servers, as studied in [21], [35]. For instance, simply increasing the queue size—such as by increasing the number of threads in each component microservice—might help delay or prevent queue overflows and dropped requests, as discussed in this paper. However, the key challenge is determining the appropriate queue size for each component microservice under dynamic workloads. This is especially true for microservices that receive multiple ingress traffic streams from various upstream services, as illustrated in Fig. 2a the `composePost` service. Too large a queue size can lead to performance issues, which have been extensively examined in previous research [24], [34], [38]. Additionally, prior studies [14], [32] have explored how different queue management strategies (e.g., priority queues) can mitigate the queuing delays. While we believe intelligent queue management strategies could alleviate queue propagation and overflow issues, they are unlikely to completely eliminate the problem.

Asynchronous architectures of microservices. The use of asynchronous architectures for high-performance internet servers has been extensively studied [19], [24], [29], [40]. Asynchronous servers typically employ an event-driven model, where an event loop manages multiple tasks concurrently. Instead of dedicating a separate thread to each task, the server listens for events (such as completed I/O operations or incoming network requests) and processes them as they occur. This allows a single thread or a small pool of threads to efficiently handle many requests. While conceptually straightforward, leveraging asynchronous architecture to build high-performance microservices is a complex challenge. For instance, asynchronous servers are notoriously difficult to program and debug due to the obscured control flow, which makes them more challenging to manage compared to traditional thread-based models [33], [39].

Dynamic Resource Allocation and Scaling. Large-scale web applications often use dynamic scaling strategies (e.g., Amazon Auto Scaling [7]) to achieve better load balancing and performance stability under dynamic workload [15], [25], [28]. However, millibottlenecks and transient cross-service queue propagation can easily evade current dynamic scaling techniques. This is because the control window of state-of-the-art scaling mechanisms typically operates at a minute-level granularity (e.g., Amazon CloudWatch [1], which monitors by default at one-minute intervals to avoid over-sensitivity to dynamic workloads). Millibottlenecks, with their sub-second durations, are too brief for these systems to detect and respond with scaling actions [12], [27]. As a result, while dynamic resource allocation and scaling are effective for handling gradual workload shifts, they may not adequately address response time instability caused by millibottlenecks and cross-service queue propagation in microservices.

VI. Conclusion

In this paper, we presented an empirical study on response time instability in cloud-based microservices. Our findings reveal that response times can become instable under moderate to high resource utilization. We illustrate two common causes of this instability: internal colocation interference and external bursty workloads. Internal colocation interference arises when multiple microservices co-exist on the same physical host, causing resource contention and performance degradation. External bursty workloads occur when a microservice experiences a sudden surge in requests, leading to system overload and response time spikes. Our results demonstrate that response time instability can significantly impact the reliability of microservices in the cloud. Future research will focus on developing strategies to mitigate these instabilities and enhance the reliability of cloud-based microservices.

Acknowledgment

This research has been partially funded by National Science Foundation by CISE/CNS (2026945), SaTC (2039653), SBE/HNDS (2024320) programs, and gifts, grants, or contracts from Fujitsu, and Georgia Tech Foundation through the John P. Imlay, Jr. Chair endowment. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or other funding agencies and companies mentioned above.

References

[1] "Amazon cloudwatch introduces high-resolution custom metrics and alarms," https://aws.amazon.com/about-aws/whats-new/2017/07/amazon-cloudwatch-introduces-high-resolution-custom-metrics-and-alarms/, accessed: 2017.

[2] "Apache thrift," https://thrift.apache.org/, accessed: 2022.

[3] "grpc: A high performance, open source universal rpc framework," https://grpc.io/, accessed: 2022.

[4] "The httperf http load generator," https://github.com/httperf/httperf, accessed: 2020.

[5] "Rubbos: Bulletin board benchmark," https://projects.ow2.org/view/rubbos/, accessed: 2021.

[6] "Rfc 6298: Computing tcp's retransmission timer," https://www.rfc-editor.org/rfc/rfc6298, 2011.

[7] Amazon, "Amazon auto scaling," https://aws.amazon.com/documentation/autoscaling, 2017.

[8] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 153–167.

[9] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, pp. 74–80, 2013. [Online]. Available: http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext

[10] Y. Gan, M. Liang, S. Dev, D. Lo, and C. 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*, 2021, pp. 135–151.

[11] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *ASPLOS*, 2019, pp. 3–18.

[12] X. Gu, Q. Wang, J. Liu, and J. Wei, "Grunt attack: Exploiting execution dependencies in microservices," in *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2024, pp. 115–128.

[13] X. Gu, Q. Wang, Q. Yan, J. Liu, and C. Pu, "Sync-millibottleneck attack on microservices cloud architecture," in *Proceedings of the 19th ACM ASIA Conference on Computer and Communications Security*, 2024, pp. 799–813.

[14] M. Harchol-Balter, T. Osogami, A. Scheller-Wolf, and A. Wierman, "Correction to: Multi-server queueing systems with multiple priority classes," *Queueing Systems*, vol. 99, pp. 397–398, 2021.

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

[16] A. Jindal, V. Podolskiy, and M. Gerndt, "Performance modeling for cloud microservice applications," in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, 2019, pp. 25–32.

[17] R. Kohavi, R. M. Henne, and D. Sommerfield, "Practical guide to controlled experiments on the web: listen to your customers not to the hippo," in *13th ACM SIGKDD*, 2007, pp. 959–967.

[18] R. Kohavi and R. Longbotham, "Online experiments: Lessons learned," *Computer*, vol. 40, no. 9, pp. 103–105, 2007.

[19] M. N. Krohn, E. Kohler, and M. F. Kaashoek, "Events can make sense." in *USENIX Annual Technical Conference*, 2007, pp. 87–100.

[20] Z. Li, Q. Chen, S. Xue, T. Ma, Y. Yang, Z. Song, and M. Guo, "Amoeba: Qos-awareness and reduced resource usage of microservices with serverless computing," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 399–408.

[21] G. Luan, P. Pang, Q. Chen, S. Xue, Z. Song, and M. Guo, "Online thread auto-tuning for performance improvement and resource saving," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 3746–3759, 2022.

[22] S. Luo, C. Lin, K. Ye, G. Xu, L. Zhang, G. Yang, H. Xu, and C. Xu, "Optimizing resource management for shared microservices: a scalable system design," *ACM Transactions on Computer Systems*, vol. 42, no. 1-2, pp. 1–28, 2024.

[23] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu, "Characterizing microservice dependency and performance: Alibaba trace analysis," in *Proceedings of the ACM SoCC*, 2021, pp. 412–426.

[24] D. Pariag, T. Brecht, A. Harji, P. Buhr, A. Shukla, and D. R. Cheriton, "Comparing the performance of web server architectures," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, pp. 231–243, 2007.

[25] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "{FIRM}: An intelligent fine-grained resource management framework for slo-oriented microservices," in *14th ({OSDI} 20)*, 2020, pp. 805–825.

[26] E. Rosenstein, "Cast ai analysis finds only 13 percent of provisioned cpus and 20 percent of memory is utilized," https://cast.ai/press-release/cast-ai-analysis-finds-only-13-percent-of-provisioned-cpus-and-20-percent-of-memory-is-utilized/, 2024.

[27] H. Shan, Q. Wang, and Q. Yan, "Very short intermittent ddos attacks in an unsaturated system," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2017, pp. 45–66.

[28] J. Shi, H. Zhang, Z. Tong, Q. Chen, K. Fu, and M. Guo, "Nodens: Enabling resource efficient and fast {QoS} recovery of dynamic microservice applications in datacenters," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 403–417.

[29] A. Sriraman and T. F. Wenisch, "{μTune}:{Auto-Tuned} threading for {OLDI} microservices," in *OSDI 18*, 2018, pp. 177–194.

[30] S. Suthar, "Strategies for efficient server utilization & energy savings," https://middleware.io/blog/efficient-server-utilization/, 2024.

[31] H. Tian, S. Li, A. Wang, W. Wang, T. Wu, and H. Yang, "Owl: Performance-aware scheduling for resource-efficient function-as-a-service cloud," in *Proceedings of the 13th Symposium on Cloud Computing*, 2022, pp. 78–93.

[32] Q. Wang, Y. Kanemasa, M. Kawaba, and C. Pu, "When average is not average: large response time fluctuations in n-tier systems," in *Proceedings of the 9th international conference on Autonomic computing*. ACM, 2012, pp. 33–42.

[33] Q. Wang, C.-A. Lai, Y. Kanemasa, S. Zhang, and C. Pu, "A study of long-tail latency in n-tier systems: Rpc vs. asynchronous invocations," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 207–217.

[34] Q. Wang, S. Malkowski, Y. Kanemasa, D. Jayasinghe, P. Xiong, C. Pu, M. Kawaba, and L. Harada, "The impact of soft resource allocation on n-tier application scalability," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011, pp. 1034–1045.

[35] Q. Wang, S. Zhang, Y. Kanemasa, C. Pu, B. Palanisamy, L. Harada, and M. Kawaba, "Optimizing n-tier application scalability in the cloud: A study of soft resource allocation," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, vol. 4, no. 2, pp. 1–27, 2019.

[36] C. Zaloumis, "Are your data centers keeping you from sustainability?" https://www.ibm.com/think/insights/are-your-data-centers-keeping-you-from-sustainability, 2024.

[37] S. Zhang, H. Shan, Q. Wang, J. Liu, Q. Yan, and J. Wei, "Tail amplification in n-tier systems: a study of transient cross-resource contention attacks," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 1527–1538.

[38] S. Zhang, Q. Wang, and Y. Kanemas, "Improving asynchronous invocation performance in client-server systems," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 907–917.

[39] S. Zhang, Q. Wang, Y. Kanemasa, J. Liu, and C. Pu, "Doublefacead: A new datastore driver architecture to optimize fanout query performance," in *Proceedings of the 21st International Middleware Conference*, 2020, pp. 430–444.

[40] S. Zhang, Q. Wang, Y. Kanemasa, H. Shan, and L. Hu, "The impact of event processing flow on asynchronous server efficiency," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 3, pp. 565–579, 2019.