Tales of the Tail: Past and Future

Christina Delimitrou
MIT
delimitrou@csail.mit.edu

Michael Marty
Google
mikemarty@google.com

***Abstract*—Tail latency has been the defining performance metric that interactive cloud services have had to meet from the beginning of the cloud as a computing paradigm. While there has been a wide spectrum of techniques, in hardware and software, that have improved tail latency for these applications, recent trends across the cloud system stack require revisiting them. Over the past few years, cloud hardware has become increasingly heterogeneous, and cloud software has been dominated by event-driven modular programming frameworks, as well as the proliferation of AI. To guarantee tail latency in this new landscape, several system advances are required. In this paper, we first review what tail latency means for cloud services, the key innovations that improved it in the past, the trends that require revisiting them, as well as the innovations that will be required for tail latency constraints to be met in the next generation of warehouse-scale computers.**

I. Introduction

If one took a snapshot of a warehouse-scale computer (WSC) ten years ago, they would see a relatively small number of large, complex applications running on mostly homogeneous, commodity hardware, not unlike a scaled out, more powerful version of your local desktop machine. A similar snapshot today would reveal a very different story.

While general purpose computing still has a fundamental role to play in the cloud, warehouse-scale systems today are much less homogeneous than they used to be. They also rely much less on readily available commodity equipment, which is how the cloud initially achieved its cost efficiency benefits. Instead, the end of Moore's Law has prompted much more creative, specialized, and heterogeneous solutions in cloud hardware, whether these target specific application classes or system-level tasks [1], [6], [9]. Despite the obvious performance and power benefits specialization offers, it also comes with an overhead in terms of management complexity. This is essential when it comes to warehouse-scale computers, because, unlike traditional enterprise computing, they rely on many fewer people to manage systems of much larger scales.

The software side of the WSC equation reveals a similar story of increased complexity. While traditional cloud applications were built as monolithic, standalone services, the emergence of modular, fine-grained programming frameworks like microservices and serverless, means that cloud applications today consist of complex topologies with hundreds if not thousand of components. When correctly applied, this programming model can offer benefits in productivity and deployment speed, but, as with hardware heterogeneity, it also contributes to increased operational complexity. Furthermore, these applications frameworks have emerged as a way to facilitate the deployment of complex applications, not as a way to improve their performance, and often experience performance unpredictability or resource inefficiencies [5], [9]. Beyond modular application architectures, the increased demand for powerful and interactive generative machine learning (ML) models has also drastically changed the ability of cloud providers to satisfy an application's performance objectives.

In this increasingly complex environment, meeting the *tail latency* guarantees that many cloud services have come to expect is ever more challenging. Tail latency, referring to a high percentile of the slowest user requests, is inherently a more challenging performance metric to satisfy than throughput (the number of operations executed per unit of time) or average request latency. This is both because it depends on the worst performing requests, and because of the request fanout that most cloud services employ [2], meaning that even a small number of slow machines and services can impact a large fraction of the overall load.

Tail latency being a key performance metric is also another point that differentiates cloud systems from high-performance computing or supercomputing. In the latter, applications are less interactive, and the performance metric of interest is typically throughput in instructions or tasks, as well as an entire job's execution time (though tail latency can indirectly impact throughput).

Finally, while previously a large fraction of cloud cycles went towards batch computation, e.g., data analytics, with modular event-driven programming frameworks and interactive ML becoming more prevalent, a larger fraction of CPU cycles has shifted towards low-latency computation. Therefore, it is important to revisit what optimizing for tail latency means in warehouse-scale computers today.

In the following sections we first review prior approaches to meet tail latency constraints in the first generation of warehouse-scale systems, ranging from hardware to application level techniques. We then discuss the recent hardware and software trends in the cloud, which require these techniques to be revisited, and conclude with the main design and management considerations we believe cloud engineers should focus on to meet tail latency moving forward.

II. Guaranteeing Tail Latency in Cloud 1.0

Given the importance in meeting the tail latency objectives of cloud services, there is a rich set of prior proposals that span the entire system stack, from hardware techniques, to operating systems and networking, and all the way to cluster management and application design. We briefly review these techniques below.

A. Hardware

High tail latency is the result of many factors, including resource contention, deep system stacks, load fluctuation and other sources of queue buildup.

One approach that prior work has used to reduce tail latency is through careful resource isolation and partitioning. This applies primarily to general-purpose, multi-tenant systems and extends to all shared resources, including the compute, power, cache capacity, memory and network bandwidth, and storage I/O. For example, by setting the CPU frequency such that the tail latency meets its Quality of Service (QoS) with minimum slack, the system achieves its performance objectives, without wasting unnecessary power.

Similarly, by partitioning shared resources across latency-critical applications or latency-critical and low-priority batch applications, systems ensure that tail latency is met, sometimes at the expense of the best-effort, low-priority workloads. Many different flavors exist for these systems, including inferring the correlation between tail latency and resources, predicting load fluctuations, and leveraging feedback-based controllers, analytical techniques, empirical profiling-based systems, or ML-based approaches.

Hardware acceleration, in addition to better raw performance, is also a way to improve tail latency by eliminating latency variability. The large number of levels of indirection in modern system stacks contributes to tail latency, because of the many places where queues can build up in hardware and software. Hardware acceleration typically removes some of these layers, reducing the sources of unpredictable performance. Many systems proposing hardware accelerator, e.g., for websearch [1] or ML training [6] show that in addition to better absolute performance, performance predictability also improves considerably. In other cases, special-purpose hardware is provisioned for the worst-case workload (such as the line rate of a network interface) at a more reasonable cost compared to similar provisioning with general-purpose hardware.

B. Operating Systems and Network Stack

Operating systems are responsible for contributing the tail latency, especially for applications operating at hundreds of microseconds, or single millisecond granularities. Operating systems (OS) were historically designed for longer running workloads, and hence their scheduling algorithms are not tailored to low-latency cloud services. Additionally, most operating systems are designed with compatibility in mind, allowing users to run arbitrary workloads. Prior work has addressed the first of these issues by designing custom schedulers that prioritize latency-critical over batch workloads, or bypassing the operating system kernel altogether when possible. Additionally, recent work has enabled microsecond-level thread scheduling as well as hardware-assisted user-level interrupts, which enable fast thread preemption and rescheduling, and are especially beneficial to low-latency cloud services. The issue of compatibility is often addressed by slimming down cloud OSes to only the functionality required for the hosted applications as opposed to code related to e.g., external devices and peripherals. OS images hosted in production cloud infrastructures are often much smaller compared to general purpose OSes, helping not only with performance overheads, but with cache locality as well.

Similarly to operating systems, the traditional networking stack adds considerable performance overheads, especially in terms of tail latency due to the multiple locations where queueing can build up, e.g., at the physical, protocol, and RPC layers. Prior work has addressed networking overhead through hardware acceleration, which executes packet processing directly in hardware, by dedicating a number of CPU cores to interrupt handling alone, or by implementing dataplane operating systems for networking that minimize data copying, fragmented request execution, and extensive request batching. Similarly, rate limiting, load shedding, or backpressure elimination techniques greatly improve tail latency, allowing cloud systems to increase their utilization without sacrificing performance.

C. Deployment and Resource Management

Cloud deployment and cluster management can dramatically impact an interactive application's tail latency. This is because cluster managers determine what type of machines an application will run on, how multi-tenancy is managed, and how many resources each task is allocated. Incorrectly managing any of these scheduling aspects can greatly degrade not only tail latency but average performance as well.

For this reason, there is a wide spectrum of proposals to improve tail latency at the cluster management level. For example, techniques like hedged and tied requests [2] sacrifice some resource efficiency for performance by scheduling and in some cases executing the same request on two different machines and using the results of the best-performing copy. Similarly, managing background activities, such as garbage collection and log compaction can avoid negatively impacting a high-priority service's latency. Along the same lines, selectively replicating frequently-accessed data more than less-frequently accessed partitions can avoid load imbalance across machines. Finally, the cluster manager placing underperforming machines "on probation" until their performance recovers is an effective way to avoid increasing tail latency because of the a small number of slow servers.

In addition to these techniques, a lot of prior work has proposed more automated approaches for cluster management that determine a latency-critical application's resource needs better than what a user can do. The motivation behind this approach is that users tend to be overly conservative, often trading off reduced resource usage for better performance because of the difficulty in precisely determining the resource needs of their applications. Automating this process removes this burden from the user and allows the system to learn an

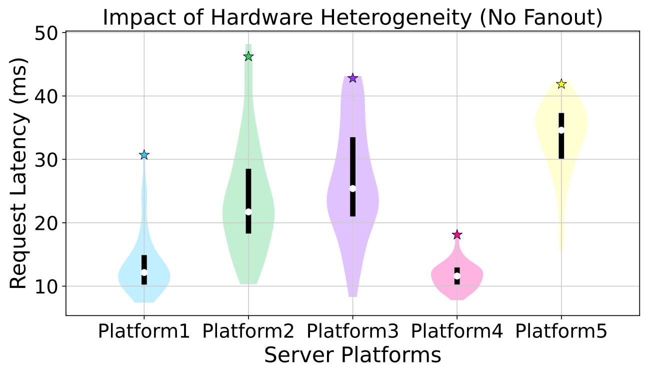

Fig. 1: Latency distribution for webserving tasks running on five different server platforms. Each violin plot shows the PDF of latency, the white dot shows the median, the black bars show the 25th and 75th percentiles, and the star shows the 99th percentile of latency.

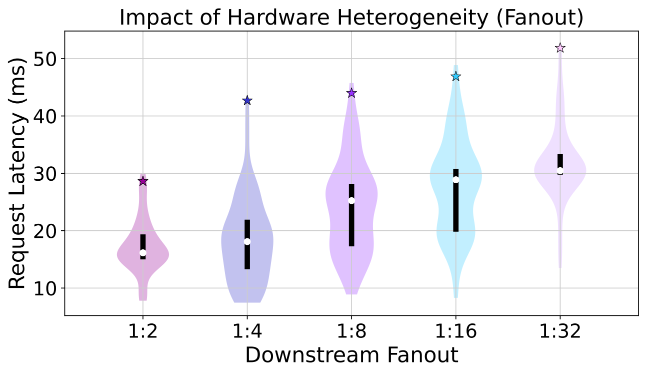

Fig. 2: Latency distribution of webserving requests as the downstream request fanout increases. Each plot shows the PDF of latency, the white dot shows the median, the black bars show the 25th and 75th percentiles, and the star shows the 99th percentile of latency.

application's resource requirements based on its similarities with the large corpus of applications the system has previously seen.

Finally, there is a line of work that focuses on reducing the overheads of deployment and cluster management. This can be done through more scalable, distributed scheduler designs that reduce scheduling latency without compromising scheduling quality, as well as reducing overheads associated with deployment, such as container instantiation. The latter is especially relevant in the case of serverless programming frameworks, where *cold start* overheads, i.e., the time required to set up a container when a new task arrives, account for a significant part of the tasks' overall execution. Optimizing these overheads is done by anticipating task arrivals, checkpointing container images in memory, or accelerating the decompression of container image snapshots using hardware support.

D. Application Design

Finally, the design of a cloud application itself can greatly impact its tail latency. From the choice of programming language, threading model, service granularity, communication framework, logic implementation, down to the tuning of operations like garbage collection, all these design choices impact performance. Prior work has addressed several of these issues. For example, systems like uTune provide a runtime that automatically selects the right threading model for a service mesh that allows an application to achieve its tail latency requirements. Similarly other proposals automatically tune the many parameters of communication frameworks like HTTPS and RPCs, time garbage collection in distributed services in a way that minimizes its latency impact, and leverage program synthesis to automatically select the right granularity for a complex service mesh that balances modularity with communication overheads.

As both low-latency ML interference and modular programming frameworks like microservices are becoming more widespread, these techniques will need to be revisited for their unique requirements.

III. Recent Cloud Trends and Tail Latency

A. Cloud Hardware

Warehouse-scale computers started out as scaled-out versions of enterprise computing, relying almost entirely on commodity, general-purpose equipment, and, as a result, achieving much better cost efficiency compared to high performance computing or supercomputers, with employed specialized architectures. The use of general-purpose infrastructure meant that programmability, even for non-expert users, was easier, lowering the bar of entry to cloud computing. It also simplified other aspects of management, including scheduling, capacity planning, and orchestration, since all servers could be simplistically assumed to have equal capabilities.

Even in this first phase of the cloud, homogeneity was relative. The building of a warehouse-scale computer is provisioned for ten to fifteen years, while the individual servers will only last three to five years, either upgraded to a later generation or replaced due to failures. Additionally, servers within the same WSC are provisioned for different applications, some optimizing for compute, while others optimize for storage or network resources. This means that at any point in time a WSC will have several different server platforms, achieving significantly different tail latencies for the same application [4].

Figure 1 shows the impact that server heterogeneity has on tail latency. The figure shows the distribution of request latencies for thousands of web-serving tasks running on NGINX. Each violin plot shows the latency distribution for a different server platform. All platforms are general purpose, two-socket, CPU-based servers and have an equal amount of compute, memory, and network resources. Despite this, the architectural implementation and configuration of each server has a significant impact on request latency, especially when looking at the tail (denoted with a star in each violin plot). In

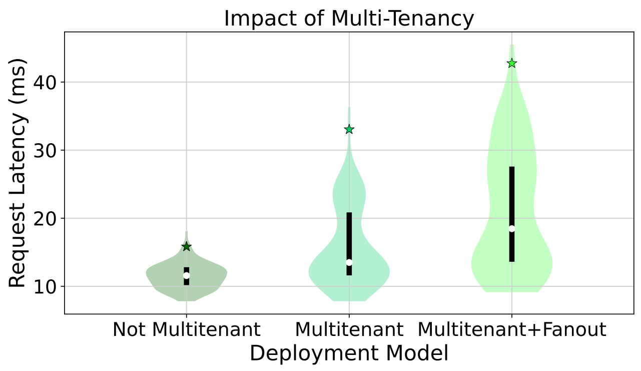

Fig. 3: Latency distribution under different deployment scenarios. From left to right, we show a non-multitenant setting, a multi-tenant cluster, and a multi-tenant cluster with request fanout. Each violin plot shows the PDF of latency, the white dot shows the median, the black bars show the 25^{th} and 75^{th} percentiles, and the star shows the 99^{th} percentile of latency.

addition to tail latency, the shape of the latency distribution varies significantly, with some servers achieving higher, but more consistent latency, while others lowering the mean but experiencing a longer tail in the distribution.

Figure 2 shows a similar experiment, while also incorporating request fanout. Now, each web-serving request is broadcast to N servers for further processing, where N is the downstream fanout. Once all the individual child requests return, the parent request aggregates their result and returns it to the client. Requests are scheduled on machines using a simple random load balancer that does not account for platform heterogeneity. As fanout increases, the probability that some child requests will be placed on a slower server also increases. Given that the aggregator needs to wait for all child requests to return before sending the response to the user, even a single slow child request can dramatically increase the end-to-end latency, especially when looking at high latency percentiles [2]. Therefore heterogeneity becomes more critical for applications with multiple tiers and request fanout.

This implicit heterogeneity has become much more definite with the recent advances in hardware acceleration for cloud services. Driven by the slowdown of scaling in process technology, many cloud providers have turned to special-purpose architectures, either built internally [6] or by third-parties, such as Intel's recent IAA accelerator engine for memory compression. Hardware acceleration falls under two categories; acceleration for specific application classes, such as deep learning [6], and acceleration for system-level tasks like networking [8], [9] or garbage collection [11], which account for a large fraction of CPU cycles in the cloud [7].

While hardware acceleration complicates cloud management and programmability, it is beneficial to tail latency for two reasons. First, by implementing logic directly in hardware, the number of layers of indirection of the system stack is reduced, and with it, a lot of performance variability they contribute to [10]. Second, by offloading computation to specialized hardware, the compute and memory pressure on the CPU is reduced. As a result, contention between different processes, which contributes to destructive resource interference also becomes more manageable. Key in the latter is the way accelerators are connected to the host CPU. Recent cache-coherent interfaces, like CXL, significantly reduce contention between CPU and accelerators by offloading data transfer to the hardware coherence protocol. To this end, a lot of prior work has shown that hardware accelerators do not only improve raw performance but performance predictability as well [1], [9].

Hardware acceleration becomes a problem for tail latency when accelerators are either scarce, of different generations or configurations, or of varied capabilities in general. In that case, prioritizing which tasks can leverage acceleration introduces performance variability, especially in applications with high request fanout.

B. Cloud Deployment

While ML training is increasingly using specialized hardware, elsewhere we see a continued shift of workloads from dedicated hardware to shared cloud computing infrastructure. This includes organizations moving from on-premise hardware to the cloud, including multi-cloud deployments, as well as web providers (such as Google) that increasingly eschew dedicated machines for latency-sensitive workloads and instead rely on improvements to cluster operating systems like Borg to reduce some sources of interference-based tail latency and to leverage tail-tolerance techniques. Multi-tenancy is essential, as it significantly increases datacenter utilization, which has traditionally remained low for latency-critical workloads. Given that most of the cost of building and operating a WSC goes towards populating it with servers, keeping those servers highly utilized is imperative.

Finally services themselves are increasingly "multi-tenant" to reduce the overheads of static resource provisioning (i.e., VMs for each user or database) and to provide the serverless illusion to users. One technique uses emerging FaaS infrastructure (functions-as-a-service), which essentially relies on the infrastructure provider to implement fine-grained resource management and scheduling. Other services implement their own fine-grained mechanisms, such as Dremel and BigQuery.

While multi-tenancy is beneficial to resource efficiency, it can be detrimental to performance. Applications sharing resources can interfere with each other, hurting each other's tail latency. A lot of prior work has focused on scheduling and resource isolation techniques to mitigate some of these overheads.

Figure 3 shows the impact of multi-tenancy on the latency distribution of the same web-serving tasks we previously discussed. The leftmost plot shows the distribution of request latency in a dedicated cluster, where requests are not sharing resources. The second plot shows the latency distribution when in the same cluster there are additional application running. While the web-serving tasks are receiving the same resources as before, their performance is worse, due to interference. Finally, the rightmost plot shows the latency distribution when the multi-tenant deployment is hosting the web-serving requests with fanout. In this case, the impact of interference is more pronounced, as even a small number of slow requests can impact a large fraction of the workload.

C. Cloud Applications

On the workload front, perhaps the largest emerging workload is the training and serving of increasingly powerful Machine Learning (ML) models. While earlier approaches to large-scale training relied on asynchronous stochastic gradient descent [3], with tail tolerant update of parameters by each worker, modern approaches rely on specialized hardware with synchronous parameter updates that essentially require a global barrier on each training step. Global barriers are at odds with tail tolerance and techniques such as backup workers can substantially increase cost for ML training workloads. For this demanding workload, we see novel hardware techniques, such as optical-circuit switching [6] to improve performance and reduce the impacts of sharing-induced latency.

In addition to the emergence of generative AI, the last few years have seen a fairly drastic change in how cloud applications are designed. In place of conventional monolithic designs, where the entire service's functionality would be included in a single codebase, compiling down to a single binary, complex topologies of multi-tier applications have emerged [5]. Serverless compute and event-driven microservices are the two driving frameworks at the forefront of this trend.

While often used interchangeably, serverless and microservices have some important differences. Serverless is an event-driven deployment strategy that simplifies cloud management, and is well suited for intermittent, data parallel, and short-lived applications will little state. It is prone to unpredictable performance due to cold starts, communication, and control plane overheads. Microservices, on the other hand, when used moderately, promote modularity, development productivity and elasticity. At the same time, because different microservices run in their own containers, possibly on separate physical nodes, they also contribute to increased network traffic, contention from multi-tenancy, and backpressure from dependencies between service tiers. All these implications are especially detrimental to tail latency, as even a small amount of network contention or queue build up can significantly degrade QoS [5].

Finally, the fact that a single service is replaced by a complex graph of–in some cases–hundreds of smaller, single-concerned services, means that the tail latency requirements for each of these services are going to be much stricter than for the original end-to-end application.

Figure 4 shows the latency distribution of a social network service implemented as a monolith and as different graphs of microservices. In all cases, the functionality of the service as far as the end user is concerned is exactly the same. We compare three different microservice topologies that use

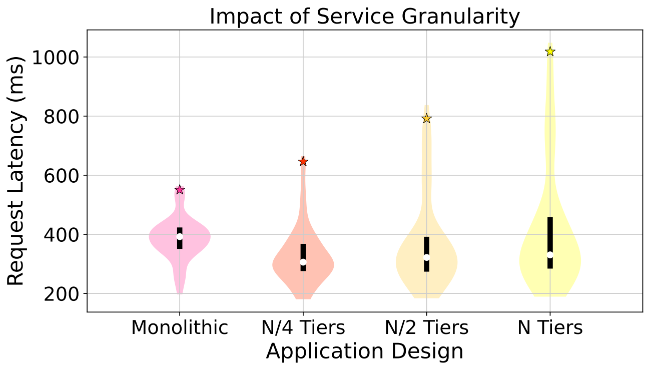

Fig. 4: Latency distribution as the granularity of services changes. The left most plot shows latency for a monolithic social network application, and the next three plots show latency for the same social network application implemented as a graph of $N/4$, $N/2$, and N microservices. Each violin plot shows the PDF of latency, the white dot shows the median, the black bars show the 25th and 75th percentiles, and the star shows the 99th percentile of latency. When moving from the monolithic application to microservices, the mean latency improves but the tail becomes progressively worse.

$N/4$, $N/2$, and N microservices, with N being the most fine-grained, to explore how service granularity impacts tail latency. All service designs are driven by the exact same traffic, and are running in a homogeneous, dedicated cluster, to avoid other sources of performance unpredictability. While the monolithic application has a higher mean latency than the microservice designs, its tail latency is better. This is because it is subject to fewer levels of queueing compared to microservices, where communication is over the network. Increasing the number of tiers in the microservice topology (making the granularity finer) does not have a significant impact on mean latency but significantly increases the tail. This is a similar finding to what happens for high-fanout services. The more tiers exist on the critical path, the higher the chance that one of them will underperform and impact the end-to-end latency.

IV. Guaranteeing Tail Latency In Cloud 2.0

The techniques described in Section II were instrumental in achieving the tail latency objectives of cloud services so far. They are not, however, enough to bring us to the next generation of warehouse-scale systems, given the trends we just discussed.

We believe that the ability to meet tail latency constraints in the next cloud generation hinges on tackling the following challenges.

A. Observability

Modern systems are deep stacks consisting of many hardware and software layers. Each layer in this stack introduces its own sources of unpredictable performance, which make guaranteeing tail latency challenging.

Observability, i.e., the ability to reason about where latency and latency variability are coming from in a complex system is imperative to correctly diagnose performance issues in cloud settings. Tracing and monitoring systems already exist in warehouse-scale computers, and usually capture both low-level resource usageand high-level performance metrics [12], such as throughput and the breakdown of latency across application tiers. When it comes to monitoring infrastructure that runs at cloud scale, there are clear trade-offs between the resolution of the information captured and the overhead capturing this information has on the system. For example, distributed tracing uses aggressive sampling of requests to avoid degrading the system's performance and resource usage, and even so, still runs into scalability bottlenecks for high-fanout services. At the same time, latency unpredictability is by its nature often attributed to short, bursty events, which can be missed when the sparsity of tracing data is high.

Moving forward, it will be critical to extend the visibility of these systems into the sources of unpredictable latency across hardware and software without sacrificing the scalability of the tracing itself. Not only will this provide a better understanding on how different layers of the system stack contribute to tail latency, but it will also simplify performance debugging efforts, whether empirical or automated.

The latter, i.e., using automation to parse complex traces and diagnose performance issues without a human in the loop, has already shown promising results in the context of cloud systems, and will only become more essential as their complexity continues to grow.

B. Shaving overheads

A lot of the latency variability in systems today comes from the many levels of indirection across the system stack. For the past few decades this complexity was sidestepped because the hardware continued to scale. As Moore's Law slows down, that complexity can no longer be ignored. Optimizing across layers, e.g., by bypassing the OS kernel or networking stack, whenever possible has already been shown to be effective for low latency services. Further reducing these overheads through vertical design, microsecond-level scheduling that is more tightly integrated with networking, more lightweight cluster management, and specialization of the system stack will be essential to meet tail latency constraints in the next generation of cloud systems.

C. Specialization

Specialization to date has been primarily driven by the compute and memory needs of a few dominating applications, such as ML. While there is certainly value in special-purpose systems for specific application domains, a different type of specialization can help with improving latency predictability.

Currently, CPUs are responsible for running, in addition to the threads serving application logic, threads executing system-level tasks, such as networking, garbage collection, memory copy, etc. These tasks have been shown to consume a large fraction of warehouse-scale CPU cycles [7]. Moreover, CPUs are not well-suited for this computation, are burdened with the overhead of the OS and networking stack, and are prone to introducing performance noise and interference in low-latency applications.

Specialization that targets this type of computation, has the potential to not only improve raw performance but reduce latency variability and free up CPU resources for applications that can benefit from them.

D. Automation

Despite their scale and complexity, warehouse-scale systems still use empirical techniques for many design and management decisions, which require a high level of human expertise. Given the trends we discussed in this paper, leveraging some degree of automation across the system stack can provide a more scalable and accurate alternative.

Naturally, as with every application of automation, especially machine learning, there is a question of explainability. Namely, if it is hard for system engineers to get any actionable insight from the model's output that can help them improve the system's design or management, the value of automation reduces. That is even more the case when the model's answer is incorrect. While this is certainly not an issue limited to applying machine learning to systems problems, it can be a roadblock for its adoption, given the SLOs these applications must meet. Moving forward, focusing on explainable or interpretable machine learning for systems will be key to harnessing the benefits that these techniques can offer for the cloud.

V. Conclusions

Tail latency has been the defining performance metric that interactive cloud services have had to meet from the beginning of the cloud as a computing paradigm. In this paper we discussed how prior work approached tail latency optimizations across the system stack, what recent trends in cloud hardware, deployment, and application design mean for tail latency, as well as what are the main challenges system architects will need to address to meet the tail latency constraints of cloud applications in the next generation of warehouse-scale computers.

References

[1] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *MICRO*, 2016.

[2] J. Dean and L. A. Barroso, "The tail at scale," in *CACM, Vol. 56 No. 2*.

[3] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks," in *NIPS*, 2012.

[4] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters," in *Proceedings of the Eighteenth ASPLOS*. Houston, TX, USA, 2013.

[5] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, “An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems,” in *Proceedings of the Twenty Fourth ASPLOS*, April 2019.

[6] N. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai, N. Patil, S. Subramanian, A. Swing, B. Towles, C. Young, X. Zhou, Z. Zhou, and D. A. Patterson, “Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings,” in *Proceedings of the 50th ISCA*, 2023.

[7] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, “Profiling a warehouse-scale computer,” *Proceedings of ISCA*, Jun. 2015.

[8] S. Karandikar, C. Leary, C. Kennelly, J. Zhao, D. Parimi, B. Nikolic, K. Asanovic, and P. Ranganathan, “A hardware accelerator for protocol buffers,” in *Proceedings of MICRO*, 2021.

[9] N. Lazarev, N. Adit, S. Xiang, Z. Zhang, and C. Delimitrou, “Dagger: Towards Efficient RPCs in Cloud Microservices with Near-Memory Reconfigurable NICs,” in *Proceedings of the Twenty Sixth ASPLOS*, April 2021.

[10] C. E. Leiserson, N. C. Thompson, J. S. Emer, B. C. Kuszmaul, B. W. Lampson, D. Sanchez, and T. B. Schardl, “There’s plenty of room at the top: What will drive computer performance after moore’s law?” *Science*, vol. 368, no. 6495, 2020.

[11] M. Maas, K. Asanović, and J. Kubiatowicz, “A hardware accelerator for tracing garbage collection,” in *Proceedings of ISCA*, 2018.

[12] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, “Dapper, a large-scale distributed systems tracing infrastructure,” Google, Inc., Tech. Rep., 2010.