Practical and Scalable ML-Driven Cloud Performance Debugging with Sage

Yu Gan
Cornell University
yg397@cornell.edu

Mingyu Liang
Cornell University
ml2585@cornell.edu

Sundar Dev
Google
sundarjdev@google.com

David Lo
Google
davidlo@google.com

Christina Delimitrou
Cornell University
delimitrou@cornell.edu

***Abstract*—Cloud applications are increasingly shifting from large monolithic services to complex graphs of loosely-coupled microservices. Despite their benefits, microservices are prone to cascading performance issues, and can lead to prolonged periods of degraded performance.**

We present Sage, a machine learning-driven root cause analysis system for interactive cloud microservices that is both accurate and practical. We show that Sage correctly identifies the root causes of performance issues across a diverse set of microservices and takes action to address them, leading to more predictable, performant, and efficient cloud systems.

I. Introduction

Cloud computing now hosts applications from practically every domain of human activity, by enabling *resource flexibility*, *cost efficiency*, and *fast deployment* [1], [4], [3]. To meet these goals, cloud services today are increasingly adopting fine-grained, modular, and event-driven programming models.

In place of large *monolithic* services that implement the entire functionality in a single binary, cloud applications now consist of hundreds or thousands of single-purpose and loosely-coupled *microservices* [8], [12]. There are several reasons making microservices appealing, including the fact that they accelerate and facilitate development, allow more agile elasticity, and enable software heterogeneity, only requiring a common API for inter-microservice communication.

At the same time, microservices introduce new system challenges. They especially complicate resource management, as dependencies between tiers introduce backpressure, causing poor performance to propagate through the system [8], [9], [12], [10]. Diagnosing such performance issues empirically is both cumbersome and error-prone, especially as typical deployments include hundreds or thousands of unique microservices.

Machine learning-based approaches have been effectively applied to cluster management for batch or single-tier interactive applications in prior work [3], [4], [6], [5]. On the performance debugging front, there has been increased attention on trace-based methods to analyze, diagnose, and even anticipate [9] performance issues in cloud services. The most closely-related previous work, Seer, leverages supervised learning to anticipate Quality-of-Service (QoS) violations, and to adjust the resource allocations to avoid them. Despite its high accuracy, Seer requires offline and online trace labeling, as well as considerable kernel-level instrumentation and fine-grained tracing to track where queues build up across the system stack. In a production environment this is non-trivial, as it involves injecting resource contention in live applications, impacting performance.

In our article published at ASPLOS'21 [7], we presented Sage, a root cause analysis system which enables practical ML-driven performance debugging entirely based on unsupervised learning, making it applicable to production environments, which cannot afford fine-grained instrumentation and invasive training. Sage uses two techniques, Causal Bayesian Networks (CBN) and Graphical Variational Autoencoders (GVAE). The CBN captures the dependencies between the microservices in an end-to-end topology, and the GVAE generates counterfactuals—hypothetical scenarios created by adjusting the state of one or more microservices—to examine the impact of microservices on end-to-end performance.

Sage does not rely on trace labeling, hence it is entirely transparent to both cloud users and application developers, scales well with the number of microservices and servers, and only relies on lightweight tracing with no application changes or kernel instrumentation. Sage targets performance issues caused by deployment, configuration, and resource provisioning reasons, as opposed to design bugs.

We have evaluated Sage both on dedicated local clusters and large cluster settings on Google Compute Platform (GCP) with several end-to-end microservices [8], and showed that it correctly identifies the microservice(s) and system resources initiating a QoS violation in over 93% of cases, and improves performance predictability without sacrificing resource efficiency.

II. System Design and Implementation

Prior work has highlighted the potential of using machine learning (ML) in cloud performance debugging. However, such techniques rely exclusively on supervised models, which require injecting resource contention on active services to correctly label the training dataset with the root causes of QoS violations [9]. This is problematic in practice, as it disrupts live applications. Additionally, prior work requires high tracing frequency and heavy instrumentation to collect metrics like the queue depth across the system stack, which is not practical in a production environment.

Sage adheres to the following design principles, which allow it to be practical, without sacrificing root cause detection accuracy:

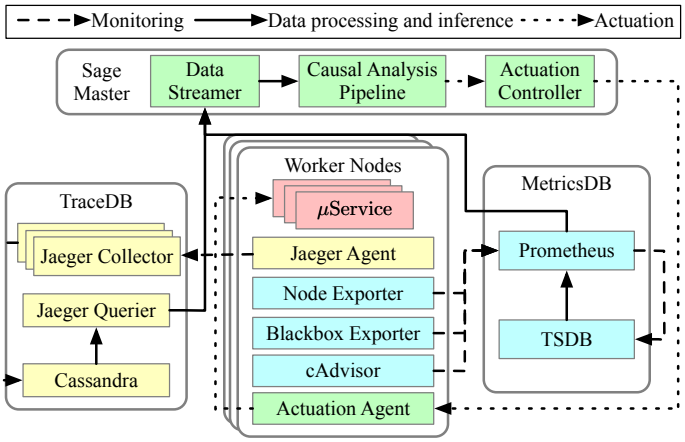

Fig. 1: Overview of Sage's system design. Sage includes a data streamer to collect and pre-process traces, a Graphical Variational Auto-Encoder (GVAE) to explore possible root causes, and an actuation controller to take the required actions to restore performance.

- **Unsupervised learning**: Sage does not require labeling training data, using instead entirely unsupervised learning techniques, trained on low-frequency traces collected during live traffic with monitoring systems readily available in most cloud providers.
- **Robustness**: Sage does not require tracking individual requests to detect temporal patterns, making it robust to tracing frequency. This is important, as production tracing systems like Dapper use aggressive sampling to reduce overheads.
- **User-level tracing**: Sage only uses user-level metrics, easily obtained through cloud monitoring APIs and service-level traces from distributed tracing frameworks, such as Jaeger. It does not require any kernel-level information, which is expensive, or even inaccessible in cloud platforms.
- **Low-overhead retraining**: A major premise of microservices is enabling frequent updates. Retraining the entire system every time for each small change is prohibitively expensive. Instead Sage implements partial and incremental retraining, whereby only the microservice that changed and its immediate neighbors are retrained.
- **Fast resolution**: Empirically examining sources of poor performance is costly in time and resources, especially given the ingest delay cloud systems have in consuming monitoring data, causing a change to take time before propagating on recorded traces. Sage models different probable root causes concurrently, restoring QoS faster.

Fig. 1 shows an overview of Sage. The system uses both distributed, RPC-level tracing for end-to-end latency monitoring, and per-node monitoring frameworks to collect hardware/OS metrics, container-level performance metrics, and network latencies. Upon detecting a QoS violation, Sage uses a generative model (GVAE) to explore the likely root causes, identify the most likely to have caused the performance issue, and take action to restore performance.

A. Tracing System

Sage includes RPC-level latency tracing and container/node-level usage monitoring. The RPC tracing system is based on Jaeger, an open-source framework, similar to Dapper and Zipkin, and augmented with the Opentracing client library, to add microservice spans and inject span context to each RPC. It measures each RPC's client- and server-side latency, and the network latency of each request and response. To avoid instrumenting the kernel to measure network latency, we use a set of probing requests to measure the heartbeat latency, and infer the request/response network delay.

We deploy one Jaeger agent per node to retrieve spans for resident microservices. We additionally enable sampling to reduce tracing overheads, and verify that with 1% sampling frequency, the tracing overhead is approximately 2.6% on the 99th percentile latency and 0.66% on the max throughput under QoS. We also ensure that sampling does not lower Sage's accuracy. To account for fluctuations in load, Sage adjusts the sampling and inference frequency to keep its detection accuracy above a configurable threshold, without introducing high overheads.

The per-node performance and usage metrics are collected using Prometheus, a widely-used open-source monitoring platform. More specifically, we deploy node, Blackbox, and cAdvisor exporters per node to measure the hardware and system metrics, network latency, and container resource usage respectively. Each metric's timeseries is stored in a centralized Prometheus TSDB. The overhead of Prometheus is negligible for all studied applications when collecting metrics every 10 seconds.

B. ML Pipeline for Root Cause Analysis

Fig. 2 shows an overview of Sage's ML pipeline. Sage relies on two techniques; first, it automatically captures the dependencies between microservices using a Causal Bayesian Network (CBN) trained on RPC-level distributed traces. The CBN also captures the latency propagation from the backend to the frontend. Second, Sage uses a generative model—a graphical variational auto-encoder (GVAE)—to generate *counterfactuals*, hypothetical scenarios which tweak the performance and/or usage of individual microservices, and infers whether the change restores QoS. Using these two techniques, Sage determines which set of microservices initiated a QoS violation, and adjusts their deployment or resource allocation.

Sage first uses the RPC-level traces collected with Jaeger to infer the microservice topology without requiring the user or cloud operator to specify it. The insight for this is that topologies change frequently, and users may not have that information or may provide it incorrectly. Sage uses these traces to build a Causal Bayesian Network (CBN) that captures how latency propagates from the back-end to front-end tiers. The CBN includes three types of nodes; the X metrics nodes, which correspond to observed resource usage metrics for each microservice and network channel, the Y latency

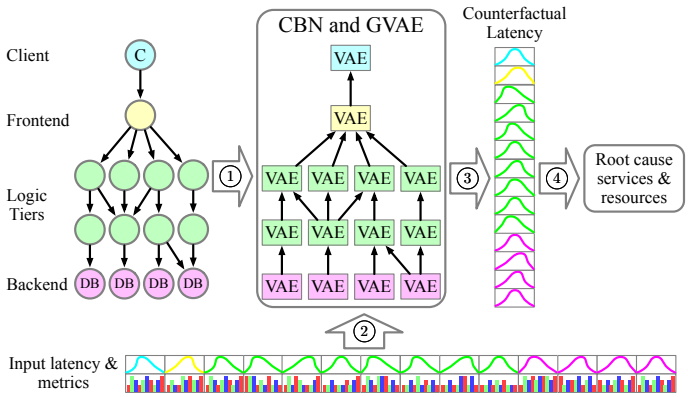

Fig. 2: Sage's ML pipeline. ①: Build Causal Bayesian Network (CBN) and Graphical Variational Auto-Encoder (GVAE). ②: Process per-tier latency and usage. ③: Generate counterfactuals with GVAE. ④: Identify root cause services & resources.

nodes, which correspond to the latency of each microservice and RPC request, and the Z latent nodes, which capture the unobservable factors responsible for latency stochasticity.

Once the CBN is created, Sage uses it to evaluate the causal relationships within and across microservices when there is a QoS violation. In a typical cloud environment, site reliability engineers (SREs) verify if a suspected root cause is correct by reverting a microservice's version or configuration to a state known to be safe, while keeping all other factors unchanged, and verifying whether QoS is restored. The disadvantage of this process is that interventions take time, and incorrect root cause assumptions hurt performance and resource efficiency. This is especially cumbersome when scaling microservices, spawning new instances, or migrating existing ones. Sage uses counterfactuals to diagnose the root cause of a QoS violation, where each counterfactual is a "suspected root cause", created using a generative model instead of actually intervening to the system. These counterfactuals help determine causality by asking what the outcome would be if the state of a microservice had been different.

Sage leverages historical tracing data to generate realistic counterfactuals. There are two challenges in this. First, the exact situation that is causing the QoS violation now may not have occurred in the past. Second, the model needs to account for the latent variables which also contribute to the variability of latency. We use a generative model to learn the latent variable and latency distributions, and use them to generate counterfactuals. We then use these counterfactuals to conduct "but-for" tests for each service and resource, and discover their causal relationship with the QoS violation. If, after intervening in the counterfactual world, the probability of meeting QoS is high, the intervened metrics have likely caused the violation.

Sage implements a two-level approach to locate a root cause, to remain lightweight and practical at scale. It first identifies the microservice(s) that caused a QoS violation, and then the specific resource(s) within a given microservice.

C. Actuation

Once Sage determines the root cause of a QoS violation, it takes action. Sage has an actuation controller in the master and one actuation agent per node. The actuation controller locates the nodes with the problematic microservices using service discovery in the container manager, and notifies their respective actuation agents to intervene. Sage focuses on deployment, configuration, and resource provisioning related performance issues, as opposed to design bugs. Therefore, once it identifies a problematic microservice, it also tries to identify the resource that caused the QoS violation. Depending on the type of resource, the actuation agent dynamically adjusts the CPU frequency, scale up/out the microservice, limit the number of co-scheduled tasks, partition the last level cache (LLC) with Intel Cache Allocation Technology (CAT), or partition the network bandwidth with the Linux traffic control's queueing discipline. The actuation agent first tries to resolve the issue by only adjusting resources on the offending node, and only when that is insufficient it scales out the problematic microservice on new nodes.

D. Handling Microservice Updates

A major advantage of microservices is that developers can easily update existing services or add new ones without impacting the entire service architecture. Sage's root cause detection accuracy can be impacted by changes to application design and deployment. Training the complete model from scratch for clusters with hundreds of nodes takes tens of minutes to hours, and is impractical at runtime. To adapt to frequent microservice changes, Sage implements *selective partial retraining* and *incremental retraining* with a dynamically reshapable GVAE, which piggybacks on the VAE's ability to be decomposed per microservice, using the CBN.

On the one hand, with selective partial retraining, we only retrain neurons corresponding to the updated nodes and their descendents in the CBN, because the causal relationships guarantee that all other nodes are not affected. On the other hand, with incremental retraining, we initialize the network parameters to those of the previous model, while adding/removing/reshaping the corresponding networks if microservices are added/dropped/updated.

If the update does not change the RPC graph or the performance and usage metrics, Sage does not retrain the model. If the update does not change the RPC graph, but the latency and usage change, Sage retrains the CVAEs of the updated microservice and its upstream microservices. The CBN remains unchanged. If the update changes the RPC graph, Sage updates the CBN. It then updates the corresponding neurons in the GVAE. Since the downstream services are not affected by the update, Sage only incrementally and partially retrains the updated microservice and its upstream microservices. For example, if a new microservice B is added between existing services A (upstream) and C (downstream), neurons would be

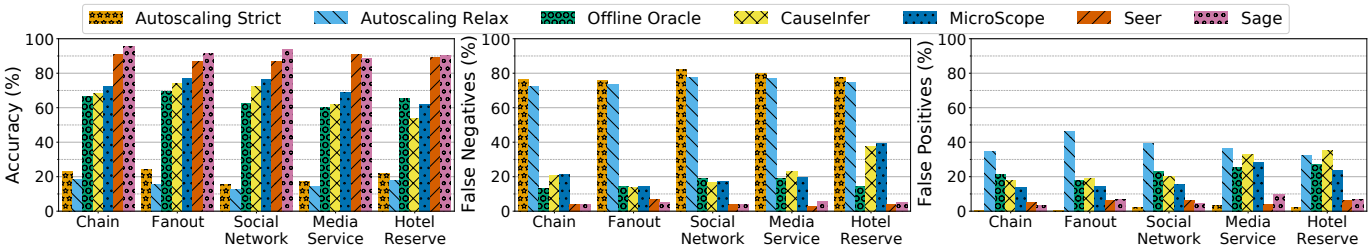

Fig. 3: Detection accuracy, false negatives, and false positives with Sage, and a number of related performance debugging and root cause analysis systems, across the two synthetic workloads, and the three end-to-end applications.

introduced for B in the corresponding networks, and only A's parameters would be retrained.

The combination of these two *transfer learning* approaches allows the model to re-converge faster, reducing the retraining time by more than $10\times$, especially when there is large fanout in the RPC graph.

III. Evaluation

A. Methodology

We evaluate Sage with five microservice topologies. We generate two synthetic microservice graphs representing common topologies; *Chain*, which includes ten serially-chained tiers, and *Fanout*, which consists of an aggregator listening to ten, fanned-out leaf microservices. In addition, we choose three end-to-end applications from DeathStarBench [8]; *Social Network*, *Media Service* and *Hotel Reservation*.

We use wrk2, an open-loop HTTP workload generator, to send requests to the web server in all five applications. To verify the ground truth for Sage's validation, we use `stress-ng` and `tc-netem` to inject CPU-, memory-, disk-, and network-intensive microbenchmarks to different, randomly-chosen microservices, to introduce unpredictable performance. Apart from resource interference, we also introduce software bugs, including concurrency bugs and insufficient threads and connections in the thread/connection pools.

We compare Sage with autoscaling techniques, which are widely used in industry, as well as recent work on performance debugging (CauseInfer [2], Microscope [11], and Seer [9]), targeting both monolithic and microservice applications.

B. Sage Validation

1) Counterfactual Generation Accuracy: We first validate the GVAE's accuracy in generating counterfactuals from the recorded latencies in the local cluster. Appropriate counterfactuals should follow the latency distribution in the training set, but also capture events that are possible, but have not necessarily happened in the past to ensure a high coverage of the performance space. There is no overlap between training and testing sets.

We examine the coefficient of determination (R^2) of the GVAE in reconstructing latencies in the test dataset. R^2 measures a model's goodness-of-fit. The closer to 1 R^2 is the more accurate the predictions. Across all five applications, R^2 values are above 0.91, denoting that the GVAE accurately reproduces the distribution and magnitude of observed latencies in its counterfactuals.

2) Root Cause Diagnosis: Fig. 3 shows Sage's accuracy in detecting root causes, compared to two autoscaling techniques, an Oracle that sets upper thresholds for each tier and metric offline, CauseInfer [2], Microscope [11], and Seer [9]. *Autoscale Strict* upscales allocations when a tier's CPU utilization exceeds 50%, and *Autoscale Relax* when it exceeds 70% (on par with AWS's autoscaling policy). Root causes include both resource-related issues—by injecting contentious kernels in a randomly-selected subset of microservices—and software bugs. Since none of the methods do code-level bug inspection, a software bug is counted as correctly-identified if the system identifies the problematic microservice correctly.

Sage significantly outperforms the two autoscalers and the offline Oracle, by learning the impact of microservice dependencies, instead of memorizing per-tier/metric thresholds for a particular system state. Similarly, Sage's false negatives and false positives are marginal. False negatives hurt performance, by misidentifying a root cause, while false positives hurt resource efficiency, by giving more resources to the wrong microservice. The 3-4% of false negatives in Sage always correspond to cases where the performance of multiple microservices was concurrently impacted by independent events, e.g., a network-intensive co-scheduled job impacted one microservice, while a CPU-intensive task impacted another. While Sage can locate multiple root causes, it takes longer, and is prone to higher errors than when a single tier is the culprit. The 3-5% of false positives are caused by spurious correlations between tiers that were not critical enough to violate QoS. In general, accuracy varies little between the five services, showing the generality of Sage across service topologies.

In comparison, the two autoscaling systems misidentify the majority of root causes; this is primarily because high utilization does not necessarily imply that a tier is the culprit of unpredictable performance. Especially when using blocking connections, e.g., with HTTP1.1, bottlenecks in one tier can backpressure its upstream services, increasing their utilization. Autoscaling misidentifies such highly-used tiers as the culprit, even though the bottleneck is elsewhere. Additionally, using a global CPU utilization threshold for autoscaling does not work well for microservices, as their resource needs vary considerably, and even lightly-utilized services can cause performance

Non-instrumented tiers	Social Network		Media Service		Hotel Reservation	
	Sage	Seer	Sage	Seer	Sage	Seer
5%	94%	90%	89%	91%	90%	89%
10%	94%	74%	89%	88%	90%	83%
20%	94%	66%	89%	74%	90%	58%
50%	94%	34%	89%	47%	90%	42%

Fig. 4: Accuracy with incomplete instrumentation for Sage and Seer. Incomplete instrumentation refers to the number of outstanding requests, which Seer uses to infer root causes, missing for a subset of randomly-selected microservices. When a large fraction of microservices cannot be instrumented, Seer's accuracy drops. Both Seer and Sage still collect per-tier latencies, and end-to-end throughput and latency.

issues. Similarly, the offline Oracle has lower accuracy than Sage, since it only memorizes per-tier thresholds for a given cluster state, and cannot adapt to changing circumstances. It also does not account for tier dependencies, or diversify between backpressure and true resource saturation.

CauseInfer and Microscope have similar accuracy since they both rely on the PC-algorithm to construct a completed, partially directed, acyclic graph (CPDAG) for causal inference. Due to statistical errors and data discretization in computing the conditional cross-entropy needed for the conditional independence test, the CPDAG's structure has inaccuracies, resulting in incorrect paths when traversing the graph to identify root causes. In contrast, Sage's CBN is directly built from the RPC graph, and considers the usage metrics of different tiers jointly, instead of in isolation, leading to much higher accuracy.

Finally, Sage and Seer have comparable accuracy and false negatives/positives; the difference lies in Sage's practicality. Unlike Seer, which requires expensive and invasive instrumentation to track the queue lengths across the system stack, and additionally relies on trace labeling to learn the QoS violation root causes, Sage only relies on sparse and non-invasive tracing, already available in most cloud providers. Sage does not require any changes in the existing application or system stack, and only relies on live data to learn root causes. This makes Sage more practical for datacenter-scale deployments, especially when the application includes libraries or tiers that cannot be instrumented. We have verified that Sage is not sensitive to the tracing frequency.

To highlight this, in Table 4 we show how Seer and Sage's accuracy is impacted from incomplete instrumentation. For Social Network, we assume that a progressively larger fraction of randomly-selected microservices cannot be instrumented. Both Sage and Seer can still track the latency, resource usage—and for Seer, the number of outstanding requests—at the "borders" (entry and exit points) of such microservices, but cannot inject any additional instrumentation points, e.g., to track the queue lengths in the OS, libraries, or application layer. Even for a small number of non-instrumented microservices, Seer's accuracy drops rapidly, as queues are misrepresented, and root causes cannot be accurately detected. In contract, Sage's accuracy is not impacted, since the system does not require any instrumentation of a tier's internal mechanics.

C. Actuation

Fig. 5a shows the tail latency for Social Network managed by Sage, the offline Oracle, Autoscale Strict (the best of the two autoscaling schemes), CauseInfer, and Microscope. We run the Social Network for 100 minutes, and inject different contentious kernels to multiple randomly-selected microservices.

Sage identifies all root causes and resources correctly. Upon detection, it notifies the actuation manager to scale up/out the corresponding resources of problematic microservices. Inference takes a few tens of milliseconds, and actuation takes tens of milliseconds to several seconds, depending on whether the adjustment is local, or requires spinning up new containers. In both cases, the process is much faster than the 30-second data sampling interval. After corrective action is applied the built-up queues start draining; latency always recovers at most after two sampling intervals from the QoS violation.

On the other hand, the offline Oracle fails to discover the problematic microservices, or takes several intervals to locate the root cause, overprovisioning resources of non-bottlenecked services in the meantime. Recovery here takes much longer, with tail latency significantly exceeding QoS. Even when the root cause is correctly identified, Oracle often overprovisions microservices directly adjacent to the culprit, as they likely exceed their thresholds due to backpressure, leading to inefficiency. The autoscaler only relies on resource utilization, and hence fails to identify the culprits in most cases. CauseInfer and Microscope similarly do not detect several root causes correctly, due to misidentifying dependencies between tiers, and lead to prolonged QoS violations. Seer is omitted as it behaves similarly to Sage.

D. Sage Retraining

We now examine Sage's real-time detection accuracy for Social Network, when microservices are updated. We roll out six updates, which include adding, updating, and removing microservices.

Each update is indicated by red dash lines labeled with A-F in Figure 5b. In A, we add a new child service to `compose-post`, close to the front-end, which processes and ranks hashtags. In B, we increase the computation complexity of `hashtag-service` by $5x$. In C, we remove the `hashtag-service`. In D, we add a new `url-preprocessing` microservice closer to the backend, between `url-shorten` and `url-shorten-mongodb`. The further downstream a new service is, the more CVAEs will have to be updated. In E, we re-incorporate the `hashtag-service`, slow down `url-preprocessing`, and remove `user-timeline` to capture Sage's behavior under multiple concurrent changes. In F, we revert `url-preprocessing` and `hashtag-service` to

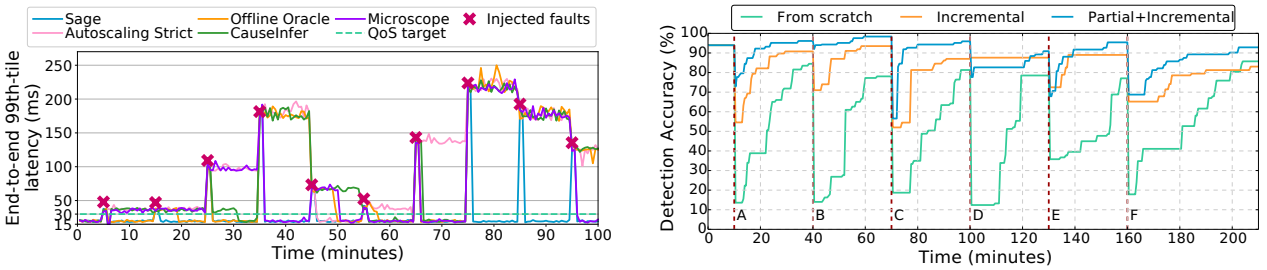

Fig. 5: (a) End-to-end tail latency for Social Network when we inject several sources of unpredictable performance to cause QoS violations. We compare Sage to CauseInfer, MicroScope, an Offline Oracle, and a conservative Autoscaling policy. (b) Detection accuracy for Sage, without and with partial & incremental retraining. Dash lines show when application updates are rolled out for the Social Network.

their previous configurations, add `user-timeline`, remove `home-timeline` and `home-timeline-redis`, and increase the CPU and memory requirements of `compose-post`.

We intentionally create significant changes in the microservice graph, and compare the accuracy of three retraining policies. *Retraining from scratch* creates a new model every time there is a change, with all network parameters re-initialized. *Incremental retraining* reuses the network parameters from the previous model, if possible, and retrains the entire network. *Partial+incremental retraining* reuses the existing network parameters and only retrains the neurons that are impacted by the updates. All approaches are trained in parallel; a new data batch arrives every 30s.

1) Retraining Time: Retraining for *partial+incremental retraining* takes a few seconds and up to a few minutes for the largest data batches. This is $3-30\times$ faster than the other two policies, because it only retrains neurons directly affected by the update, a much smaller set compared to the entire network. The more microservices are updated, and the deeper the updated microservices are located in the RPC dependency graph (updates D, E, F), the higher the retraining time.

2) Root Cause Detection Accuracy: Fig. 5b shows that *partial+incremental retraining* and *incremental retraining* have the lowest accuracy drop immediately after an update. On the contrary, *retraining from scratch* loses its inference ability right after an update, since the network parameters are completely re-initialized, and the model forgets its prior knowledge. Note that the previous model cannot be used after the update, because introducing a new microservice changes the GVAE and network dimensions. *Partial+incremental retraining* converges much faster than the other two models, because of its shorter retraining time, which prevents neurons irrelevant to the service update from overfitting to the small training set and forgetting the previously learned information.

E. Sage Scalability

Finally, we deploy the Social Network on 188 containers on GCE using Docker Swarm. We replicate all stateless tiers on 2-10 instances each, depending on their resource needs, and shard the caches and databases.

We use two Intel Xeon 6152 processors with 44 cores for training and inference. Sage takes 124 min to train from scratch on the local cluster and 148 min on GCE. Root cause inference takes 49ms on the local cluster and 62ms on GCE. The root cause detection accuracy is unchanged. Although we deploy $6.7\times$ more containers on GCE, the training and inference times only increase by 19.4% and 26.5% respectively. Sage's good scalability is primarily due to the system collecting a percentile tensor of latency and usage metrics across all per-tier replicas, and avoiding high-frequency, detailed tracing for root cause detection.

IV. Discussion

Sage leverages causal models and unsupervised learning to identify the culprits of unpredictable performance in complex microservice topologies, making it practical for large-scale production systems. Sage addresses the challenges this problem presents through three main contributions.

Enabling practical, ML-driven performance debugging: Sage highlights the need for ML-driven performance debugging in emerging cloud programming frameworks, like microservices. Unlike traditional cloud applications where manual performance debugging was an option, the complexity and dependencies of microservice deployments make automated solutions a necessity. Given this, it is critical to identify the ML techniques that can be applied in a production environment, offer high resolution accuracy, and act transparently to users.

Sage specifically shows that unsupervised learning, an approach usually shied away from for cloud performance debugging, can be effectively applied in realistic scenarios, improving the system's performance predictability and resource efficiency with much less tracing information than required by previous systems. Not only does this make ML-driven performance debugging practical for private large-scale deployments, but it can also be effective in public cloud systems where the provider does not always have access to a third party application's source code. This opens up a new research direction for ML-driven performance debugging, which is not only applicable to microservices, but other complex environments as well, such as HPC and edge swarms. Similarly, Sage

enables ML-driven debugging that goes beyond provisioning-related issues and also applies to deployment configuration, correctness, and even security concerns.

Finally, Sage is specifically designed with frequent application updates in mind; a major premise of microservices. Unlike previous systems which need to be retrained from scratch to recover their high root cause detection accuracy, Sage embeds the RPC dependency graph in its inference process, only requiring the model components impacted by the change to be retrained, without a loss in accuracy.

The more complex modern systems become, the more critical it will be for practical ML-driven systems, like Sage, to help their design and management.

Evaluation using realistic end-to-end microservices: All experiments in Sage are using realistic, end-to-end applications built with microservices, including social networks and media services from the DeathStarBench suite [8]. These applications are built using APIs and services often found in production systems, including the Thrift RPC framework, NGINX, Memcached, MongoDB, and Redis, improving the evaluation's representativeness. Additionally, the applications used in Sage's evaluation are open-source, enabling researchers in both academia and industry to reproduce our results, and build upon our system.

Promoting explainability in ML for systems: A recurring issue with applying ML to systems is that ML is often treated as a black box, offering little insight into how its output can be used to improve the system's design or management. In Sage we have specifically applied techniques whose output can be interpreted into actionable decisions. For example, Sage has helped us identify tiers with excessive resource utilization, prompting a redesign with alternate API libraries, tiers with high communication between microservices, prompting a merge into a single tier, and tiers whose logging infrastructure was significantly contributing to QoS violations, resulting in a redesign of their tracing systems. As ML is further integrated in complex systems, it is imperative to design techniques that not only are accurate but can offer useful design and management insights to both platform architects and application developers.

Acknowledgements

We sincerely thank the anonymous reviewers for their feedback on earlier versions of this manuscript. This work was in part supported by NSF CAREER Award CCF-1846046, NSF grant NeTS CSR-1704742, two Google Faculty Research Awards, a Sloan Foundation Research Scholarship, a Microsoft Research Fellowship, an Intel Faculty Rising Star Award, a Facebook Research Award, and a John and Norma Balen Sesquicentennial Faculty Fellowship.

Yu Gan is a Ph.D. candidate in the School of Electrical and Computer Engineering at Cornell University. He works on cloud computing, computer architecture, and root cause analysis for cloud microservices. He is a student member of IEEE and ACM. Contact him at yg397@cornell.edu.

Mingyu Liang is a Ph.D. candidate in the School of Electrical and Computer Engineering at Cornell University. He works on computer architecture, cloud computing, and new cloud programming models. He is a student member of IEEE and ACM. Contact him at ml2585@cornell.edu.

Sundar Dev is a Performance Engineer at Google where he works on improving performance, efficiency, and reliability of the large-scale distributed computing infrastructure that is used by all of Google's user facing services. Contact him at sundarjdev@google.com.

David Lo is a Performance Engineer at Google, where he leads a team that works on optimizing the performance and efficiency of Google's planet-scale computing infrastructure. Lo received a Ph.D. degree in electrical engineering from Stanford University. He is a member of IEEE and ACM. Contact him at davidlo@google.com

Christina Delimitrou is an Assistant Professor with the School of Electrical and Computer Engineering, Cornell University, where she works on computer architecture and distributed systems. Delimitrou received a Ph.D. degree in electrical engineering from Stanford University. She is a member of IEEE and ACM. Contact her at delimitrou@cornell.edu.

References

[1] L. Barroso and U. Hoelzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. MC Publishers, 2009.

[2] P. Chen, Y. Qi, P. Zheng, and D. Hou, "Causeinfer: Automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems," in *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, 2014, pp. 1887–1895.

[3] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Houston, TX, USA, 2013.

[4] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-Efficient and QoS-Aware Cluster Management," in *Proceedings of the Nineteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Salt Lake City, UT, USA, 2014.

[5] C. Delimitrou and C. Kozyrakis, "Bolt: I Know What You Did Last Summer... In The Cloud," in *Proc. of the Twenty Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[6] C. Delimitrou, D. Sanchez, and C. Kozyrakis, "Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters," in *Proceedings of the Sixth ACM Symposium on Cloud Computing (SOCC)*, August 2015.

[7] 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*, ser. ASPLOS 2021. New York, NY, USA: Association for Computing Machinery, April 2021, p. 135–151. [Online]. Available: https://doi.org/10.1145/3445814.3446700

[8] 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 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2019.

[9] Y. Gan, Y. Zhang, K. Hu, Y. He, M. Pancholi, D. Cheng, and C. 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 (ASPLOS)*, April 2019.

[10] 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 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2021.

[11] J. Lin, P. Chen, and Z. Zheng, "Microscope: Pinpoint performance issues with causal graphs in micro-service environments," in *International Conference on Service-Oriented Computing*. Springer, 2018, pp. 3–20.

[12] Y. Zhang, W. Hua, Z. Zhou, E. Suh, and C. Delimitrou, "Sinan: ML-Based and QoS-Aware Resource Management for Cloud Microservices," in *Proceedings of the Twenty Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2021.