Sage: Practical and Scalable ML-Driven Performance Debugging in Microservices

Yu Gan*
yg397@cornell.edu
Cornell University
Ithaca, New York, USA

Mingyu Liang
ml2585@cornell.edu
Cornell University
Ithaca, New York, USA

Sundar Dev
sundarjdev@google.com
Google
Sunnyvale, California, USA

David Lo
davidlo@google.com
Google
Sunnyvale, California, USA

Christina Delimitrou
delimitrou@cornell.edu
Cornell University
Ithaca, New York, USA

ABSTRACT

Cloud applications are increasingly shifting from large monolithic services to complex graphs of loosely-coupled microservices. Despite the advantages of modularity and elasticity microservices offer, they also complicate cluster management and performance debugging, as dependencies between tiers introduce backpressure and cascading QoS violations. Prior work on performance debugging for cloud services either relies on empirical techniques, or uses supervised learning to diagnose the root causes of performance issues, which requires significant application instrumentation, and is difficult to deploy in practice.

We present *Sage*, a machine learning-driven root cause analysis system for interactive cloud microservices that focuses on practicality and scalability. Sage leverages unsupervised ML models to circumvent the overhead of trace labeling, captures the impact of dependencies between microservices to determine the root cause of unpredictable performance online, and applies corrective actions to recover a cloud service's QoS. In experiments on both dedicated local clusters and large clusters on Google Compute Engine we show that Sage consistently achieves over 93% accuracy in correctly identifying the root cause of QoS violations, and improves performance predictability.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; **n-tier architectures**; • **Software and its engineering** → *Software performance*; • **Computing methodologies** → *Causal reasoning and diagnostics*; Neural networks.

KEYWORDS

cloud computing, microservices, performance debugging, QoS, counterfactual, Bayesian network, variational autoencoder

ACM Reference Format:
Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 2021. Sage: Practical and Scalable ML-Driven Performance Debugging in Microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3445814.3446700

1 INTRODUCTION

Cloud computing has reached proliferation by offering *resource flexibility*, *cost efficiency*, and *fast deployment* [20, 25, 37–43, 52, 77]. As the scale and complexity of cloud services increased, their design started undergoing a major shift.

In place of large *monolithic* services that encompassed the entire functionality in a single binary, cloud applications have progressively adopted fine-grained modularity, consisting of hundreds or thousands of single-purpose and loosely-coupled *microservices* [2, 17, 18, 47–49, 104, 108]. This shift is increasingly pervasive, with cloud-based services, such as Amazon, Twitter, Netflix, and eBay, having already adopted this application model [2, 17, 18]. There are several reasons that make microservices appealing, including the fact that they accelerate and facilitate development, they promote elasticity, and enable software heterogeneity, only requiring a common API for inter-microservice communication.

Despite their advantages, microservices also introduce new system challenges. They especially complicate resource management, as dependencies between tiers introduce backpressure effects, causing unpredictable performance to propagate through the system [48, 49]. Diagnosing such performance issues empirically is both cumbersome and prone to errors, especially as typical microservices deployments include hundreds or thousands of unique tiers. Similarly, current cluster managers [29, 38, 41, 44, 70, 72, 73, 75, 77, 82, 83, 86, 95, 99, 112, 115] are not expressive enough to account for the impact of microservice dependencies, thus putting more pressure on the need for automated root cause analysis systems.

Machine learning-based approaches have been effective in cluster management for batch applications [36], and for batch and interactive, single-tier services [38, 41]. On the performance debugging front, there has been increased attention on trace-based methods to

*This work was not done at Google.

ASPLOS '21, April 19–23, 2021, Virtual, USA

ACM ISBN 978-1-4503-8317-2/21/04…$15.00
https://doi.org/10.1145/3445814.3446700

analyze [30, 46, 85], diagnose [19, 23, 32, 35, 54, 60, 63, 81, 91, 110, 113, 114], and in some cases anticipate [47, 49, 109] performance issues in cloud services. While most such systems target cloud applications, the only one focusing on microservices is Seer [49]. Seer leverages a deep learning model to anticipate upcoming QoS violations, and adjusts the resources per microservice to avoid them. Despite its high accuracy, Seer uses supervised learning, which requires offline and online trace labeling, as well as considerable kernel-level instrumentation and fine-grained tracing to track the number of outstanding requests across the system stack. In a production system this is non-trivial, as it involves injecting resource contention in live applications, which can impact performance and user experience.

We present Sage, a root cause analysis system that leverages unsupervised learning to identify the culprit of unpredictable performance in complex graphs of microservices in a scalable and practical manner. Specifically, Sage uses Causal Bayesian Networks to capture the dependencies between the microservices in an end-to-end application topology, and counterfactuals (events that happen given certain alternative conditions in a hypothetical world) through a Graphical Variational Autoencoder to examine the impact of microservices on end-to-end performance. Sage does not rely on data labeling, hence it can be entirely transparent to both cloud users and application developers, making it practical for large-scale deployments, scales well with the number of microservices and machines, and only relies on lightweight tracing that does not require application changes or kernel instrumentation, which would be difficult to obtain in practice. 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 Engine (GCE) with several end-to-end microservices [48], and showed that it correctly identifies the microservice(s) and system resources that initiated a QoS violation in over 93% of cases, and improves performance predictability without sacrificing resource efficiency.

2 RELATED WORK

Below we review work on the system implications of microservices, cluster managers designed for multi-tier services and microservices, and systems for cloud performance debugging.

2.1 System Implications of Microservices

The increasing popularity of fine-grained modular application design, microservices being an extreme materialization of it, has yielded a large amount of prior work on representative benchmark suites and studies on their characteristics [48, 55, 104]. μSuite [104] is an open-source multi-tier application benchmark suite containing several online data-intensive (OLDI) services, such as image similarity search, key-value stores, set intersections, and recommendation systems. DeathStarBench [48] presents five end-to-end interactive applications built with microservices, leveraging Apache Thrift [1], Spring Framework [12], and gRPC [5]. The services implement popular cloud applications, like social networks, e-commerce sites, and movie reviewing services. DeathStarBench also explores the hardware/software implications of microservices, including their resource bottlenecks, OS/networking overheads, cluster management challenges, and sensitivity to performance unpredictability. Accelerometer [105] characterizes the system overheads of several Facebook microservices, including I/O processing, logging, and compression. They also build an analytical model to predict the potential speedup of a microservice from hardware acceleration.

2.2 Microservices Cluster Management

Microservices have complicated dependency graphs, strict QoS targets, and are sensitive to performance unpredictability. Recent work has started exploring the resource management challenges of microservices. Suresh et al. [108] design Wisp, a dynamic rate limiting system for microservices, which prioritizes requests in the order of their deadline expiration. uTune [107] auto-tunes the threading model of multi-tier applications to improve their end-to-end performance. GrandSLAm [66] improves the resources utilization of ML microservices by estimating the execution time of each tier, and dynamically batching and reordering requests to meet QoS. Finally, SoftSKU [106] characterizes the performance of the same Facebook microservices as [105] across hardware and software configurations, and searches for their optimal resource configurations using A/B testing in production.

2.3 Cloud Performance Debugging

There is extensive prior work on monitoring and debugging performance and efficiency issues in cloud systems. Aguilera et al. [19] built a tool to construct the causal path of a service from RPC messages without access to source code. X-Trace [46] is a tracing framework portable across protocols and software systems that detects runtime performance issues in distributed systems. It can identify faults in several scenarios, including DNS resolution and overlay networks. Mystery Machine [33] leverages a large amount of cloud traces to infer the causal relationships between requests at runtime. There are also several production-level distributed tracing systems, including Dapper [100], Zipkin [16], Jaeger[7], and Google-Wide Profiling (GWP) [90]. Dapper, Zipkin and Jaeger record RPC-level traces for sampled requests across the calling stack, while GWP monitors low-level hardware metrics. These systems aim to facilitate locating performance issues, but are not geared towards taking action to resolve them.

Autopilot [94] is an online cluster management system that adjusts the number of tasks and CPU/memory limits automatically to reduce resource slack while guaranteeing performance. Sage differs from prior work on cloud scheduling, such as [41, 50, 76, 115], in that it locates the root cause of poor performance only using the end-to-end QoS target, without explicitly requiring to define per-tier performance service level agreements (SLAs).

Root cause analysis systems for cloud applications are gaining increased attention, as the number of interactive applications continues to increase. Several of these proposals leverage statistical models to diagnose performance issues [54, 109, 113]. Cohen et al. [35] build tree-augmented Bayesian networks (TANs) to predict whether QoS will be violated, based on the correlation between performance and low-level metrics. Unfortunately, in multi-tier applications, correlation does not always imply causation, given

the existence of backpressure effects between dependent tiers. ExplainIt! [62] leverages a linear regression model to find root causes of poor performance in multi-stage data processing pipelines which optimize for throughput. While the regression model works well for batch jobs, latency is more sensitive to noise, and propagates across dependent tiers.

CauseInfer [28] as well as Microscope [71] build a causality graph using the PC-algorithm, and use it to identify root causes with different anomaly detection algorithms. As with ExplainIt!, they work well for data analytics, but would be impractical for latency-critical applications with tens of tiers, due to the high computation complexity of the PC-algorithm [65]. Finally, Seer [49] is a supervised CNN+LSTM model that anticipates QoS violations shortly before they happen. Because it is proactive, Seer can avoid poor performance altogether, however, it requires considerable kernel-level instrumentation to track the number of outstanding requests across the system stack at fine-granularity, which is not practical in large production systems. It also requires data labeling to train its model, which requires injecting QoS violations in active services. This sensitivity to tracing frequency also exists in Sieve [111], which uses the Granger causality test to determine causal relationships between tiers [21, 101].

3 ML FOR PERFORMANCE DEBUGGING

3.1 Overview

Sage is a performance debugging and root cause analysis system for large-scale cloud applications. While the design centers around interactive microservices, where dependencies between tiers further complicate debugging, Sage is also applicable to monolithic architectures. Sage diagnoses the *root cause* [57] of end-to-end QoS violations, and applies appropriate corrective action to restore performance. Fig. 1 shows an overview of Sage's ML pipeline. Sage relies on two techniques, each of which is described in detail below; first, it automatically captures the dependencies between microservices using a Causal Bayesian Network (CBN) trained on RPC-level distributed traces [16, 100]. The CBN also captures the latency propagation from the backend to the frontend. Second, Sage uses a graphical variational auto-encoder (GVAE) to generate hypothetical scenarios (counterfactuals [51, 79]), which tweak the performance and/or usage of individual microservices to values known to meet QoS, 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.

While prior work has highlighted the potential of ML for cloud performance debugging [49], such techniques rely exclusively on supervised models, which require injecting resource contention on active services to correctly label the training dataset with root causes of QoS violations [49]. This is problematic in practice, as it disrupts the performance of live services. Additionally, prior work requires high tracing frequency and heavy instrumentation to collect metrics like the number of outstanding requests across the system stack, which is not practical in a production system and can degrade performance.

Sage instead adheres to the following design principles:

- **Unsupervised learning**: Sage does not require labeling training data, and it diagnoses QoS violations using low-frequency traces collected during live traffic using tracing systems readily available in most major cloud providers.
- **Robustness to sampling frequency**: 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 [100] employ aggressive sampling to reduce overheads [34, 96]. In comparison, previous studies [49, 98, 111] collect traces at millisecond granularity, which can introduce significant overheads.
- **User-level metrics**: Sage only uses user-level metrics, easily obtained through cloud monitoring APIs and service-level traces from distributed tracing frameworks, such as Jaeger [7]. It does not require any kernel-level information, which is expensive, or even inaccessible in cloud platforms.
- **Partial retraining**: A major premise of microservices is enabling frequent updates. Retraining the entire system every time the code or deployment of a microservice changes 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 the impact of the different probable root causes concurrently, restoring QoS faster.

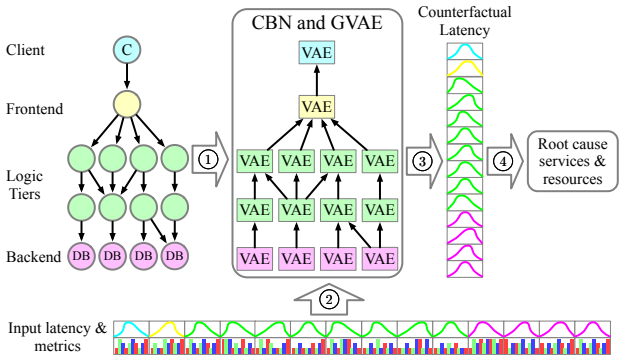

Figure 1: Sage's ML pipeline. ①: Build Causal Bayesian Network (CBN) and Graphical Variational Auto-Encoder (GVAE). ②: Process per-tier latency and usage. ③: Generate counterfactuals with GVAE. ④: Identify root cause services & resources.

3.2 Microservice Latency Propagation

3.2.1 Single RPC Latency Decomposition. Fig. 2 shows the latency decomposition of an RPC across client (sender) and server (receiver). The client initiates an RPC request via the `rpc0_request` API at ①. The request then waits in the RPC channel's send queue and gets written to the Linux network stack via the `sendmsg` syscall at ②. The packets pass through the TCP/IP protocol and are sent out from the client's NIC. They are then transmitted over the wire and switches and arrive at the server's NIC. After being processed by

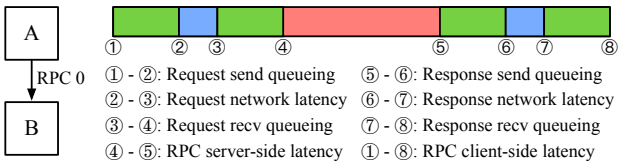

Figure 2: RPC latency breakdown. Red bars: RPC server-side latency, blue bars: network latency, green bars: application queueing.

the server's network protocol stack at ③, the request is queued in the RPC channel's receive queue, waiting to be processed via the `rpc0_handler`, which starts at time ④ and ends at ⑤. Finally, the RPC response follows the same process from server to client, until it is received by the client's application layer at time ⑧. ① - ⑧ and ④ - ⑤ are the application-level client- and server-side latencies, respectively. ② - ③ and ⑥ - ⑦ are the latencies in the network protocol, switches, and wiring. ① - ②, ③ - ④, ⑤ - ⑥, and ⑦ - ⑧ is the queueing time in the application layer of the client and server, respectively.

Timestamps for the user-level events ①, ④, ⑤, and ⑧ can be obtained with distributed tracing frameworks, such as Jaeger. Timestamping ②, ③, ⑥, and ⑦, would require probing the Linux kernel with high-overhead tools, like SystemTap [45]. Instead, we approximate the request/response network delay by measuring the latency of heartbeat signals between client and server, when queueing in the application is zero.

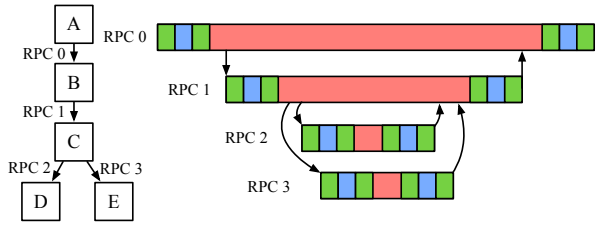

Figure 3: Dependency graph and traces of nested RPCs.

3.2.2 Markov Property of RPC Latency Propagation. Multiple RPCs form a tree of nested traces in a distributed monitoring system. Fig. 3 shows an example RPC dependency graph with five services, four RPCs, and its corresponding latency traces. When the user request arrives at *A*, it sends `RPC0` to service *B*. *B* further forwards the request to *C* via `RPC1`, and *C* sends it to the backend services *D* and *E* via `RPC2` and `RPC3` in parallel. After processing the responses from *D* and *E*, *C* replies to *B*, and *B* replies to *A*, as `RPC1` and `RPC0` return.

The server-side latency of any non-leaf RPC is determined by the processing time of the RPC itself and the waiting time (i.e., client-side latency) of its child RPCs. This latency propagates through the RPC graph to the frontend. Since the latency of a child RPC cannot propagate to its parent without impacting its own latency, the latency propagation follows a *local Markov property*, where each latency is conditionally independent on its non-descendant RPCs, given its child RPC latencies [69]. For instance, the latency

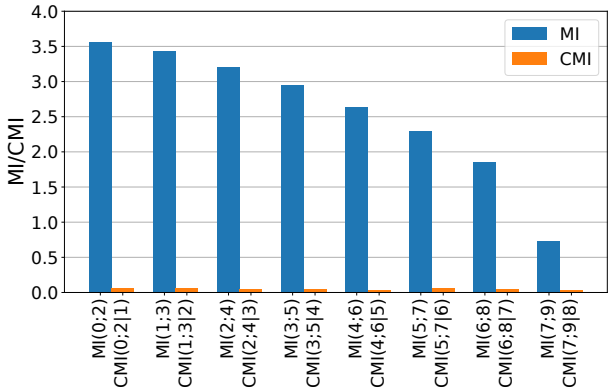

Figure 4: Mutual Information (MI) of two distance-of-2 RPCs, and Conditional Mutual Information (CMI) given the server latency of the middle RPC. CMI of zero means that the latencies of the two RPCs are conditionally independent, given the latency of their in-between RPC.

of `RPC0` is conditionally independent of `RPC2` and `RPC3`, given the latency of `RPC1`.

In information theory, *mutual information* measures the reduction of uncertainty in one random variable given another random variable. Two random variables are independent or conditionally independent if their mutual information (MI) or conditional mutual information (CMI) is zero [53]. Fig. 4 shows the MI of the server-side latencies of two RPCs with distance of two, and their CMI, given the server-side latency of the in-between RPC, in a 10-microservice chain. The MI of each two non-adjacent RPCs is blocked by the latency of the RPC in the middle, making them conditionally independent [27].

3.3 Modeling Microservice Dependency Graphs

3.3.1 Causal Bayesian Networks. A CBN is a directed acyclic graph (DAG), where the nodes are random variables and the edges indicate their conditional dependencies, from cause to effect [84, 88]. Sage uses three node types:

- **Metric nodes (X)**: They contain resource-related metrics of all services and network channels collected with tools, like Google Wide Profiling [24, 34, 96]. They are the exogenous variables that cause latency variances across RPCs, and fall into two groups: *server-* and *network-related*. Server-related metrics (X^s), include CPU utilization, memory bandwidth, context switches, etc., and impact the server's processing time. Network-related metrics (X^{net}), such as the round trip time (RTT), packet loss rate, network bandwidth, etc., affect the delay of RPC channels. The set of sufficient metrics was derived by selecting those features that improve the model's accuracy, without overfitting to a specific deployment. Features that may be capturing overlapping information are discarded by the network by demoting the corresponding neuron weights. To keep the shape of the vector for each metric the same regardless of the replicas per tier, we use a vector of percentiles [64], e.g., [10th%, ..., 90th%, 100th%] computed across the tier's replicas.

- **Latency nodes (Y)**: These include client-side latency (Y^c), server-side latency (Y^s), and request/response network delay (Y^{req} and Y^{resp}) of all RPCs of Sec. 3.2.1. We also use a vector of percentiles to represent the RPC latency distribution. Since the RPC tail latency correlates more closely with QoS, high percentiles are sampled more finely.
- **Latent variables (Z)**: These nodes contain the unobservable factors that are responsible for latency stochasticity. They are critical to generate the counterfactual latencies Sage relies on to diagnose root causes (Sec. 3.5). We divide latent variables to server-related variables (Z^s) which capture individual microservices, and network-related variables (Z^{net}), which capture links between them. The latent variables are dependent of the metric nodes.

We separate network- from server-related variables because the conditionally-independent network-related metrics we are interested in do not directly impact the server-related metrics, and vice versa. For example, high network bandwidth traffic between two tiers may be correlated with high CPU utilization of one or both tiers, but not memory bandwidth by itself, without impacting any other metric. We then construct the CBN among the three node classes for all RPCs, based on their causal relationships and latency propagation obtained via the distributed tracing system (Sec. 3.2). We use four rules to construct the CBN:

(i) Metric nodes have no causes because they are exogenous variables set outside the model. Since the distribution of a latent variable is modulated by its corresponding metric node, there is an edge from X to Z.
(ii) The server-side latency of an RPC call is determined by the client-side latency of its child RPCs (if any), and server-related metrics and latent variables of the microservice tier initiating the call.
(iii) The client-side latency of an RPC is the result of its server-side latency, request and response network delay, and the server-related metrics and latent variables of the microservice which invoked it.
(iv) The request/response network delays are defined by an RPC's network-related metrics and latent variables.

Figure 5 shows an example of the CBN of a three-microservice dependency chain. The nodes with solid lines (X and Y) are observed, while the nodes with dashed lines (Z) are latent variables that need to be inferred. The arrows in the RPC graph and CBN have opposite directions because the latency of one RPC is determined by the latency of its child RPCs.

3.3.2 Latency Distribution Factorization. We consider the microservice latencies and usage metrics in the CBN to be random and i.i.d variables from the underlying distribution. Using the CBN, we can factorize the joint distribution into the product of individual tier distributions, conditional on their parent variables. Factorization is needed to later build the graphical model of Sec. 3.5, which will explore possible root causes. We are interested in the following distributions:

- The conditional distribution of latency given the observed metrics and latent variables $P(Y \mid X, Z)$,

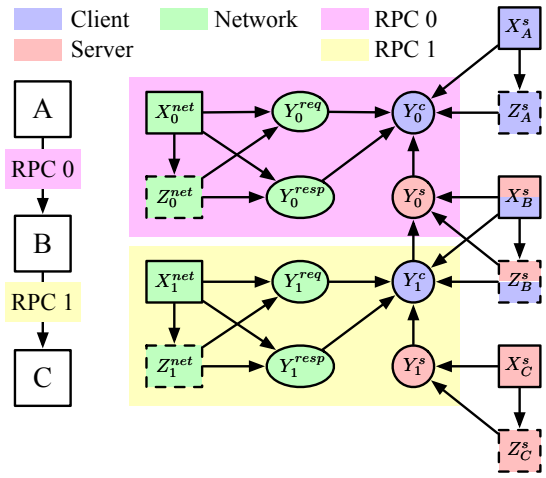

Figure 5: The RPC graph of a 3-service chain, and its corresponding Causal Bayesian Network (CBN). Blue-colored nodes correspond to client-related variables, red-colored nodes correspond to server-related variables, and green-colored nodes correspond to network-related nodes. The background color refers to the CBN nodes that correspond to a given RPC in the topology. X nodes capture resource-related metrics, Y nodes capture the server-client latencies of individual RPCs, and Z nodes correspond to the unobserved, latent variables that are responsible for the latency stochasticity.

- The prior distribution of latent variables Z given the observed metrics, $P(Z \mid X)$, and
- The posterior distribution of latent variables Z, given the observed metrics and latency values $Q(Z \mid X, Y)$.

Given the conditional independence relationship represented by the CBN, we can decompose the conditional distribution $P(Y \mid X, Z)$ as follows:

$$P(Y \mid X, Z) = \prod_{i=1}^{n} \Big[P\big(Y_i^c \mid \texttt{deDeps}(Y_i^c)\big) \cdot P\big(Y_i^s \mid \texttt{deDeps}(Y_i^s)\big) \cdot P\big(Y_i^{req} \mid \texttt{deDeps}(Y_i^{req})\big) \cdot P\big(Y_i^{resp} \mid \texttt{deDeps}(Y_i^{resp})\big) \Big], \tag{1}$$

where $\texttt{deDeps}(Y_i)$ are the dependent nodes of Y_i, which are used as the inputs of the decoders in Sec 3.5. The dependent nodes of each type of Y_i can be represented as

$$\begin{aligned} \texttt{deDeps}(Y_i^c) &= \{Y_i^{req}, Y_i^{resp}, Y_i^s, X^s_{\texttt{client}(i)}, Z^s_{\texttt{client}(i)}\}, \\ \texttt{deDeps}(Y_i^s) &= \{Y^c_{\texttt{children}(i)}, X^s_{\texttt{server}(i)}, Z^s_{\texttt{server}(i)}\}, \\ \texttt{deDeps}(Y_i^{req}) &= \{X_i^{net}, Z_i^{net}\}, \\ \texttt{deDeps}(Y_i^{resp}) &= \{X_i^{net}, Z_i^{net}\}, \end{aligned} \tag{2}$$

where $\texttt{client}(i)$ and $\texttt{server}(i)$ denote the client and server of RPC i, $\texttt{children}(i)$ are the set of child RPCs that RPC i invokes, and n in the total number of RPCs. Similarly, we can also decompose

$P(X \mid Z)$ as

$$P(Z \mid X) = \prod_{j=1}^{m} P(Z_j^s \mid X_j^s) \cdot \prod_{i=1}^{n} P(Z_i^{net} \mid X_i^{net}) \tag{3}$$

where m is the total number of microservices, and $Q(Z \mid X, Y)$ as

$$Q(Z \mid X, Y) = \prod_{j=1}^{m} P\big(Z_j^s \mid \texttt{enDeps}(Z_j^s)\big) \cdot \prod_{i=1}^{n} P\big(Z_i^{net} \mid \texttt{enDeps}(Z_i^{net})\big), \tag{4}$$

where $\texttt{enDeps}(Z_j)$ are the dependent nodes of Z_j, which are used as inputs of the encoders in Sec 3.5. They can be written as

$$\begin{aligned} \texttt{enDeps}(Z_j^s) &= \big\{X_j^s, Y^s_{\texttt{served}(j)}, Y^c_{\texttt{invoked}(j)}, Y_{\texttt{v\_structure}(j)}\big\}, \\ \texttt{enDeps}(Z_i^{net}) &= \big\{X_i^{net}, Y_i^{req}, Y_i^{resp}\big\}, \end{aligned} \tag{5}$$

where $\texttt{served}(j)$ are the set of server-side RPCs served on service j, $\texttt{invoked}(j)$ are the set of client-side RPCs invoked from service j to its downstream services, and $Y_{\texttt{v\_structure}(j)}$ includes all Y nodes forming a V-structure with Z_j^s and having both edges directed to any node in $Y^s_{\texttt{served}(j)}$ and $Y^c_{\texttt{invoked}(j)}$ (a pattern of $Y_{\texttt{v\_structure}(j)} \rightarrow \{Y \mid Y \in Y^s_{\texttt{served}(j)} \vee Y^c_{\texttt{invoked}(j)}\} \leftarrow Z_j^s$ in the CBN). Both deDeps and enDeps are derived from the information flow according to the structure of the CBN.

3.4 Counterfactual Queries

Sage uses counterfactual queries [80, 88] to diagnose the root cause of unpredictable performance. In a typical cloud environment, site reliability engineers (SREs) can verify if a suspected root cause is correct by reverting a microservice's version or resource configuration to a state known to be safe, while keeping all other factors unchanged, and verifying whether QoS is restored. Sage uses a similar process, where "suspected root causes" are generated using counterfactual queries, which determine causality by asking what the outcome would be if the state of a microservice had been different [58, 78, 80]. Such counterfactuals can be generated by adjusting problematic microservices in the system in a similar way to how SREs take action to resolve a QoS violation. The disadvantage of this is that interventions take time, and incorrect root cause assumptions hurt performance and resource efficiency. This is especially cumbersome when scaling microservices out, spawning new instances, or migrating existing ones.

Instead, 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 Z which also contribute to the distribution of Y. Therefore, we use a generative model to learn the latent distribution $P(Z \mid X)$ and the latency distribution $P(Y \mid X, Z)$, and use them to generate counterfactual latencies Y, given input metrics X. We then use the counterfactuals to conduct "but-for" tests for each service and resource, and discover their causal relationship with the QoS violation. If, after intervening, the probability of meeting QoS exceeds a threshold, the intervened metrics caused the violation.

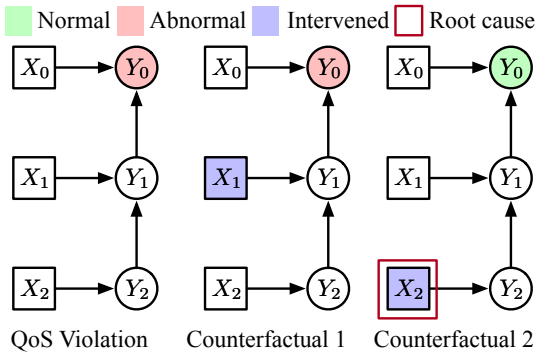

Figure 6: Detecting root causes using counterfactuals. Initially the system tries to diagnose the root cause of poor performance by reverting the CPU utilization of tier 1, X_1, to values known to meet QoS. Since this does not resolve the end-to-end QoS violation, the system then generates a counterfactual that sets the utilization of tier 2, X_2, to values known to meet QoS, which ends up resolving the QoS violation.

3.5 Generating Counterfactuals

Conditional deep generative models, such as the conditional variational autoencoders (CVAE) [103] is a common tool to generate new data from an original distribution. Generally, it infers the distribution of low-dimensional latent space variables (Z) from high-dimensional data (Y) and tags (X), and samples from that distribution to generate new data with specific tag. Recent studies have showed that these techniques can also be used to generate counterfactuals for causal inference [74].

Fig. 6 shows an example of detecting the root cause of a QoS violation in the 3-tier chain of Fig. 5. Assume that the CPU utilization of Services 1 and 2 is abnormal (different from values that meet QoS). We evaluate the hypothetical end-to-end latency of two counterfactuals; one where Service 1's utilization is normal, with all other metrics unchanged, and one where Service 2's utilization is normal. If fixing Service 1 does not restore QoS, as in *Counterfactual 1*, then Service 1 *alone* is not the root cause. If fixing the utilization of Service 2 restores QoS, as in *Counterfactual 2*, then it is the root cause. Not being enough to restore QoS does not mean that a service is not part of the problem; if single microservices do not restore QoS, Sage considers mixes of tiers.

To generate counterfactuals, we build a network of CVAEs according to the structure of the CBN. We adapt the CVAE in [103], a widely-used hybrid model with a CVAE and a Gaussian stochastic neural network (GSNN). The CVAE network can be further decomposed into an encoder, decoder, and prior network. During the training phase, the CVAE receives a mini-batch of X and Y from the training set. The encoder learns the posterior distribution of Z, given the observed X and Y ($Q_\phi(Z \mid X, Y)$), and the prior network learns the prior distribution of Z, observing only X ($P_\psi(Z \mid X)$). The decoder then reconstructs the input target Y, based on Z sampled from the posterior distribution and X, i.e., $P_\theta(Y \mid X, Z)$, where $Z \sim Q_\phi(Z \mid X, Y)$. The encoder, decoder, and prior networks are

constructed with multi-layer perceptrons (MLPs) parameterized with θ, ϕ, and ψ, respectively. During the generation phase, we use the prior network to modulate the distribution of Z, given X, and use Z sampled from that distribution together with X to generate Y. During training, we minimize the latency reconstruction loss plus a regularization term of Kullback-Leibler (KL) divergence, i.e., the negative variational lower bound [67]:

$$L_{\text{CVAE}}(X,Y,Z;\theta,\phi,\psi) = \underbrace{-\mathbb{E}_{Z\sim Q_\phi(Z|X,Y)}\big[\log P_\theta(Y \mid X,Z)\big]}_{\text{reconstruction loss}} + \beta \cdot \underbrace{D_{KL}\big(Q_\phi(Z \mid X,Y) \parallel P_\psi(Z \mid X)\big)}_{\text{KL divergence regularization}} \tag{6}$$

where $\beta > 0$ is a hyperparameter that encourages to identify disentangled latent factors in Z [56]. The reconstruction loss term allows the encoder to extract useful input features, and the decoder to accurately reconstruct the original data from the latent variables. The KL divergence regularization minimizes the difference between the posterior distribution $Q_\phi(Z \mid X,Y)$ and the prior distribution $P_\psi(Z \mid X)$. We further add a GSNN to reconstruct Y by sampling Z from the prior distribution. It tackle concerns that the CVAE alone may not be enough to train a conditional generative model, because it uses the posterior distribution from the encoder during training and the prior distribution to draw Z samples during generation [61, 103].

$$L_{\text{GSNN}}(X,Y,Z;\theta,\psi) = -\mathbb{E}_{Z\sim P_\psi(Z|X)}\big[\log P_\theta(Y \mid X,Z)\big]. \tag{7}$$

Therefore, a hybrid model that adds a GSNN can be written as

$$L_{\text{CVAE\_hybrid}}(X,Y,Z;\theta,\phi,\psi) = \alpha \cdot L_{\text{CVAE}} + (1-\alpha)\cdot L_{\text{GSNN}}, \tag{8}$$

where $\alpha \in [0,1]$ is a hyperparameter to balance the loss between two networks.

Although using a single CVAE for the entire microservice graph would be simple, it has several drawbacks. First, it lacks the CBN's structural information which is necessary to avoid ineffectual counterfactuals based on spurious correlations. Second, it prohibits partial retraining, which is essential for frequently-updated microservices. Finally, it is less explainable since it does not reveal how the latency of a problematic service propagates to the frontend. Therefore, we construct one small CVAE per microservice with few fully connected and dropout layers, and connect the different CVAEs according to the structure of the CBN to form the graphical variational autoencoder (GVAE). Because $P(Y \mid X,Z)$, $P(Z \mid X)$, and $Q(Z \mid X,Y)$ can be factorized via Eq 1, Eq 3, and Eq 4, the final loss function is:

$$L_{\text{GVAE\_hybrid}}(X,Y,Z;\theta,\phi,\psi) = \sum_{i=1}^{m}\Big[\alpha L_{\text{CVAE}_i} + (1-\alpha)L_{\text{GSNN}_i}\Big] \tag{9}$$

where CVAE_i and GSNN_i is the CVAE and the Gaussian stochastic network for service i. The encoders and prior networks are trained entirely in parallel. The decoders require the outputs of the parent decoders in the CBN as inputs, and are trained serially. The maximum depth of the CBN determines the max number of serially-cascaded decoders.

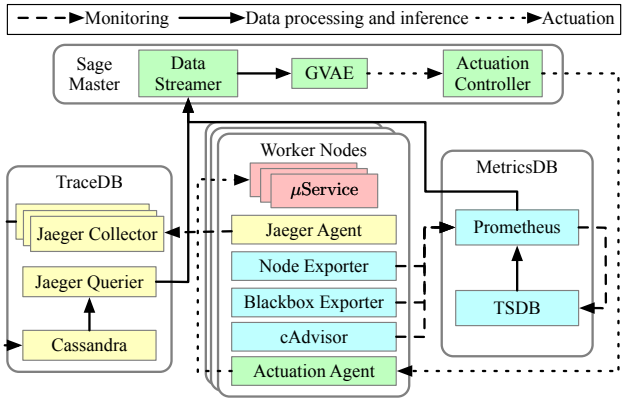

Figure 7: Overview of Sage's system design. Sage includes a data streamer, a Graphical Variational Auto-Encoder (GVAE) ML model, and an actuation controller. The data streamer collects and pre-processes collected traces and performance metrics from TraceDB and MetricsDB. The GVAE ML model predicts culprit microservices and resources, and then triggers the actuation controller to dynamically adjust the appropriate hardware resources, if one or more resources are identified are the source of unpredictable performance.

4 SAGE DESIGN

Sage is a root cause analysis system for interactive microservices. Sage relies on RPC-level tracing to compose a CBN with the microservice topology, and per-node tracing for per-tier latency distributions. Below we discuss Sage's monitoring system (Sec. 4.1), training and inference pipeline (Sec. 4.2), its actuator once a root cause has been identified (Sec. 4.3), and how Sage handles application changes (Sec. 4.4).

Fig. 7 shows an overview of Sage. The system uses Jaeger [7], a distributed RPC tracing system for end-to-end execution traces, and the Prometheus Node Exporter [11], Blackbox Exporter [10], and cAdvisor [4] to collect hardware/OS metrics, container-level performance metrics, and network latencies. Each metric's time-series is stored in the Prometheus TSDB [9, 89]. At runtime, Sage queries Jaeger and Prometheus to obtain real-time data. The GVAE then infers the root cause of any QoS violation(s), at which point Sage's actuator adjusts the offending microservice's resources.

Sage uses a centralized master for trace processing, root cause analysis, and actuation, implemented in approximately 6KLOC of Python, and per-node agents for trace collection and container deployment. It also maintains two hot stand-by copies of the master for fault tolerance. The GVAE model is built in PyTorch, with each VAE's encoder, decoder, and prior network using a DNN with 3-5 fully connected layers, depending on the input node number. We also use batch normalization between every two hidden layers for faster convergence, and a dropout layer after the last hidden layer to mitigate overfitting.

4.1 Tracing Systems

Sage includes RPC-level latency tracing and container/node-level usage monitoring. The RPC tracing system is based on Jaeger [7],

an open-source framework, similar to Dapper [100] and Zipkin [16], and augmented with the Opentracing client library [8], 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. Sage records two spans per RPC; one starts when the client sends the RPC request and ends when it receives the response, while the other starts when the server receives the RPC request and ends when it sends the response to the client, both at application level. To avoid instrumenting the kernel to measure network latency (Sec. 3.2.1), 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. The Jaeger agents flush the spans to a replicated Jaeger collector for aggregation, which stores them in a Cassandra database. 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 incurring high overheads.

The per-node performance and usage metrics are collected using Prometheus, a widely-used open-source monitoring platform [9]. More specifically, we deploy node, blackbox, and cAdvisor exporters per node to measure the hardware/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.

4.2 Root Cause Analysis

To diagnose a root cause, Sage first relies on the Data Streamer to fetch and pre-process the tracing data. The Streamer queries Jaeger and Prometheus for an interval's log data over HTTP, and preprocesses them using feature encoding, aggregation, dimensionality reduction, and normalization. It outputs RPC latency percentiles across the sampled requests, and performance/usage percentiles across the replicas of each tier.

Sage initializes and trains the GVAE model offline with all initially-available latency and usage data. It then periodically retrains the model as new requests come in [31, 59, 87, 116]. Retraining happens even when there are no application changes, to account for changes in user behavior. Sage handles design changes with partial and incremental retraining to minimize overheads and accelerate convergence (Sec. 4.4). Every time training is triggered, the GVAE streams in batches of tracing tensors to update its network parameters. Online learning models are prone to *catastrophic forgetting*, where the model forgets previous knowledge upon learning new information [68, 87]. To avoid this, we interleave the current and previous data in the training batches. Sage could also be prone to *class imbalance*, where the number of traces that meet QoS is significantly higher than those which violate QoS. In that event, the Data Streamer oversamples the minority class to create a more balanced training dataset, preventing the model from being penalized for generating counterfactuals that violate QoS.

At runtime, Sage uses the latest version of the GVAE to diagnose QoS violations. Based on training data, Sage first labels the medians of per-tier performance and usage when QoS is met as *normal values*. If during execution QoS is violated, the GVAE generates counterfactuals by replacing a microservice's performance/usage with their respective *normal values*.

Sage implements a two-level approach to locate a root cause, to remain lightweight and practical at scale. It first uses service-level counterfactuals to locate the culprit microservice that initiated the performance degradation, and then uses resource-level counterfactuals in the culprit, to identify the underlying reason for the QoS violation and correct it. More precisely, for each microservice, Sage restores all its metrics to their normal values and uses the GVAE to generate the counterfactual end-to-end latency based on the CBN structure. Since the CBN indicates the causal relationship between a given RPC and the examined microservice, for all non-causally related RPCs, the GVAE reuses their current per-tier latencies in the counterfactual. The microservice that reduces the end-to-end latency to just below QoS is signaled as the culprit. After locating the offending microservice, Sage generates resource-specific counterfactuals to examine the impact of each hardware resource on end-to-end performance. The instantaneous CPU frequency and utilization act as CPU indicators, memory utilization as a memory indicator, network bandwidth, TCP latency, and ICMP latency as network indicators, etc. Compared to a one-level approach which tries to jointly locate the service and resource, the two-level scheme is simpler and faster.

Finally, there are cases where multiple microservices are jointly responsible for a QoS violation. In such cases, the GVAE iteratively explores microservice combinations when generating counterfactuals, by adding each time the tier which would have reduced the end-to-end latency the most.

4.3 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 GVAE notifies the actuation controller, which 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 the problematic microservice or microservices, it also tries to identify the system resource that caused the QoS violation. Depending on which resource is identified as instigating the QoS violation, the actuation agent will dynamically adjust 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 moves to scale out the problematic microservice on new nodes, or migrate it, especially for stateful backends, which are almost never migrated.

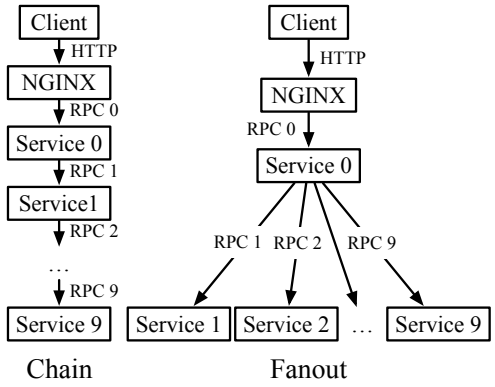

Figure 8: RPC dependency graph for the two synthetic Chain and Fanout services.

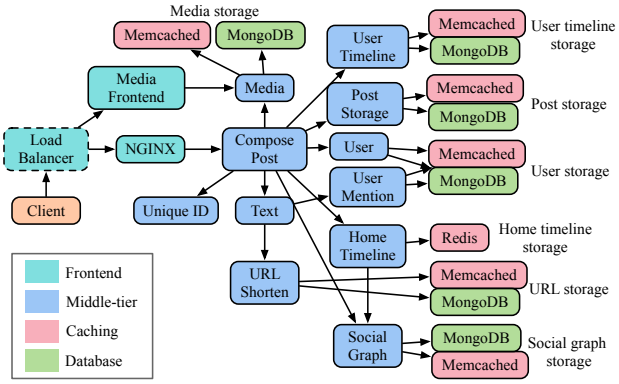

Figure 9: Social Network microservice architecture [48]. Client requests first reach a front-end load balancer, which evenly distributes them across the N webserver instances. Then, depending on the type of user request, a number of logic, mid-tiers will be invoked to create a post, read a user's timeline, follow/unfollow users, or receive recommendations on new users to follow. At the right-most of the figure, the requests reach the back-end databases, implemented both with in-memory caching tiers (memcached and Redis), and persistent databases (MongoDB).

4.4 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 ability to diagnose QoS violations can be impacted by changes to application design and deployment, such as new, updated, or removed microservices. 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 instead implements *selective partial retraining* and *incremental retraining* with a dynamically reshapable GVAE similar to [116], 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 uses the low-frequency distributed traces collected with Jaeger to update 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 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×, especially when there is large fanout in the RPC graph. To collect sufficient training data quickly after an update, we temporarily increase the tracing sampling rate until the model converges.

5 METHODOLOGY

5.1 Cloud Services

5.1.1 Generic Thrift Microservices. *Apache Thrift* [1, 102] is a scalable, widely-used RPC framework. We implement a Thrift code generator to synthesize customizable graphs of resource-intensive microservices. We can configure the number of microservices, the processing time, the RPC graph, and how RPCs interleave to emulate different functional/timing dependencies. We generate two common microservice topologies; *Chain* and *Fanout*, shown in Fig. 8.

In Chain, each microservice receives a request from its upstream service, sends the request to its downstream tier after processing, and responds to its parent once it gets the results from its child. In Fanout, the root service broadcasts requests to the leaf tiers, and returns the result to the client only after all children tiers have responded. We choose the Chain and Fanout topologies because they highlight different behaviors in terms of root cause analysis, and because most real microservice topologies are combinations of the two [48, 66, 104].

5.1.2 Social Network. End-to-end service in DeathStarBench [48] implementing a broadcast-style social network. Users can follow/unfollow other users and create posts embedded with text, media, urls, and user mentions, which are broadcast to their followers. They can also read posts, get user recommendations, and see ads. Fig. 9 shows the Social Network architecture. The backend uses Memcached and Redis for caching, and MongoDB for persistent storage. We use the socfb-Reed98 Facebook network dataset [93] as the social graph, which contains 962 users and 18.8K follow relationships.

5.1.3 Media Service. End-to-end service in DeathStarBench implementing a movie review website. Users can submit reviews and ratings of movies. They can also browse the information of movies, including their plot, photos, videos, cast, and review information. We use a subset of TMDB database which contains 1000 movies and 1000 users. Fig. 10 shows the architecture of Media Service.

5.1.4 Hotel Reservation. It is a hotel reservation website which enables users to search for hotels, place reservations and get recommendations of nearest hotels based on the users' locations. The application is implemented in Go, and the services communicate over gRPC. The dataset consists of 80 hotels and 500 users. Fig. 11 illustrates the Hotel Reservation microservice architecture.

5.2 Systems

5.2.1 Local Cluster. We use a dedicated local cluster with five 2-socket 40-core servers with 128GB RAM each, and two 2-socket 88-core servers with 188GB RAM each. Each server is connected to a 40Gbps ToR switch over 10Gbe NICs. All services are deployed as Docker containers.

5.2.2 Google Compute Engine. We also deploy the Social Network on a GCE cluster with 84 nodes in `us-central1-a` to study Sage's scalability. Each node has 4-64 cores, 4-64GB RAM and 20-128GB SSD, depending on the microservice(s) deployed on it. There is no interference from external jobs.

5.3 Training Dataset for Validation

We use wrk2 [3], an open-loop HTTP workload generator, to send requests to the web server in all three applications. To verify the ground truth for Sage's validation in Sec. 6, we use `stress-ng` [13] and `tc-netem` [14] 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 for Sage to detect, including concurrency bugs and insufficient threads and connections in the pool.

6 EVALUATION

6.1 Sage Validation

6.1.1 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) and root-mean-square error (RMSE) of the GVAE in reconstructing latencies in the test dataset. R^2 and RMSE measure a model's goodness-of-fit. The closer to 1 R^2 is, and the lower the RMSE, the more accurate the predictions. Across all three applications, R^2 values are above 0.91, and RMSEs are 7.8, 5.1, and 3.2 respectively for the Chain, Fanout and Social Network services, denoting that the GVAE accurately reproduces the distribution and magnitude of observed latencies in its counterfactuals. Note that the standard deviations of latencies in the validation set are high, highlighting that generating representative counterfactuals is non trivial.

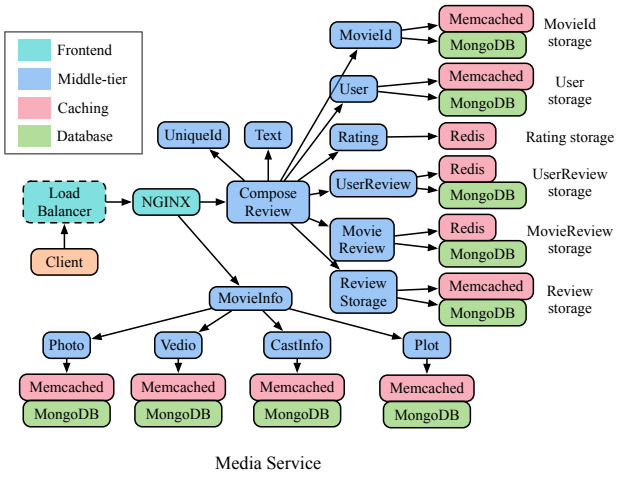

Figure 10: Media Service architecture [48]. Client requests first reach a front-end load balancer, which evenly distributes them across the N webserver instances. Then, depending on the type of user request, a number of logic, mid-tiers will be invoked to browse information about a movie, create a new movie review, or get recommendations on movies a user may enjoy. At the right-most of the figure, the requests reach the back-end databases, implemented both with in-memory caching tiers (memcached and Redis), and persistent databases (MongoDB).

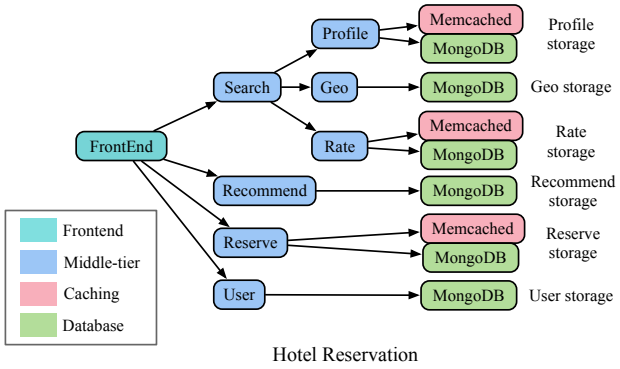

Figure 11: Hotel reservation microservice architecture [48]. Client requests first reach a front-end webserver, and, depending on the type of requests, are then directed to logic tiers implementing functionality for searching hotels, completing hotel reservations, updating a user profile, and getting recommendations on available hotels. At the right-most of the figure, the requests reach the back-end databases, implemented both with in-memory caching tiers (memcached), and persistent databases (MongoDB).

6.1.2 Root Cause Diagnosis. Fig. 12 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 [28], Microscope [71], and Seer [49]. *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-related issue is counted as correctly-identified if the system identifies the problematic microservice correctly.

Sage significantly outperforms the two autoscalers and even the offline oracle, by learning the impact of microservice dependencies, instead of memorizing per-tier/metric thresholds for a particular cluster state. Similarly, Sage's false negatives and false positives are marginal. False negatives hurt performance, by missing the true source of unpredictable performance, 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, that 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. Out of the three services, Fanout has slightly lower accuracy, due to the fact that a single misbehaving leaf can significantly impact the end-to-end performance. In general, accuracy varies little between the three services, showing the generality of Sage across service architectures.

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 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, e.g., load fluctuation, tier changes, or contentious co-scheduled tasks. It can also 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 [65] 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 from distributed traces, 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 in each microservice, and additionally relies on supervised 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 the root causes of QoS violations, instead of offline training. This makes Sage more practical and portable at 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 13 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 implementation.

6.2 Actuation

Fig. 14 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 to apply corrective action, 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. Furthermore, 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 resource inefficiency. The autoscaler only relies on resource utilization, and hence fails to identify the culprits in the majority of cases, leading to prolonged QoS violations. CauseInfer and Microscope similarly do not detect several root causes correctly,

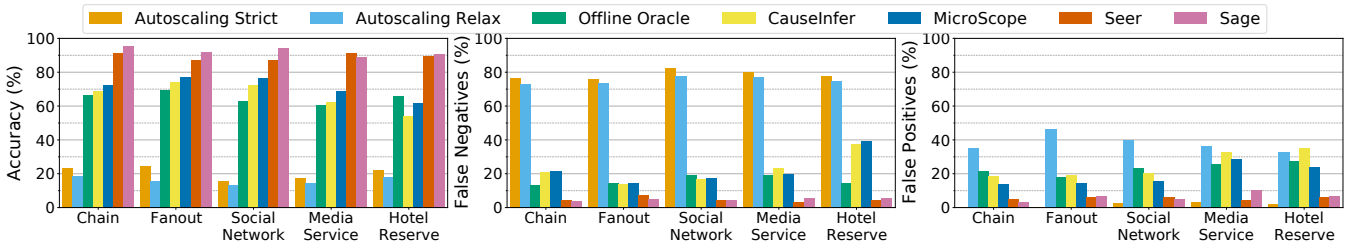

Figure 12: Detection accuracy, false negatives, and false positives with Sage, and a number of related performance debugging and root cause analysis systems, across the two synthetic workloads, and the three end-to-end applications.

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%

Figure 13: Accuracy with incomplete instrumentation with Sage and Seer, for each of the three end-to-end applications. Incomplete instrumentation refers to the number of outstanding requests, which Seer uses to infer the root cause of unpredictable performance, missing from a subset of randomly-selected microservices in the end-to-end service. When the non-instrumented tiers are off the critical path, the missing instrumentation does not significantly impact detection accuracy. When, however, a larger fraction of overall microservices cannot be instrumented, Seer's accuracy drops. Both for Seer and Sage we still collect per-tier latencies, and end-to-end throughput and latency.

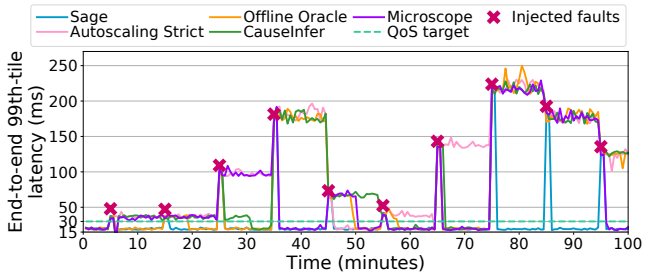

Figure 14: End-to-end tail latency for Social Network when we inject several sources of unpredictable performance to cause QoS violations. We compare Sage to CauseInfer, MicroScope, an Offline Oracle, and a conservative Autoscaling policy.

due to misidentifying dependencies between tiers, and lead to prolonged QoS violations. We omit Seer from the figure as it behaves similarly to Sage.

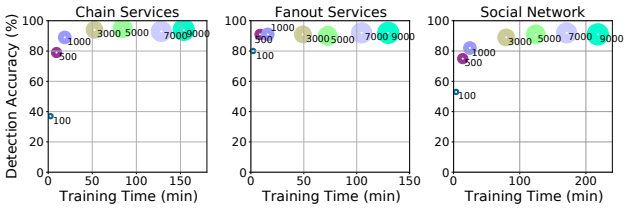

Figure 15: Sensitivity to training set size (samples) for the two synthetic services and the Social Network.

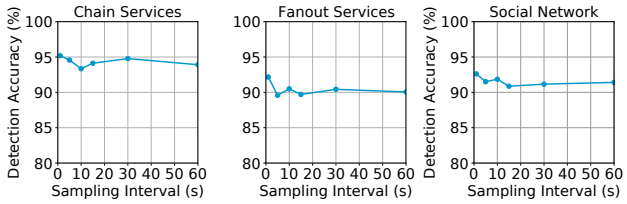

Figure 16: Sensitivity to the sampling rate for the two synthetic services and the Social Network.

6.3 Sensitivity Analysis

6.3.1 Training Data Size. Figure 15 shows the root cause detection accuracy and training time for Sage across all three applications, as we increase the size of the training dataset. The circle sizes are proportional to the sizes of the training datasets. The training data are collected on the local cluster with a sampling interval of 30 seconds, consistent with the granularity at which QoS is defined. The smallest dataset is collected in 50 minutes, and the largest in over three days. Sage's detection accuracy increases until the number of samples reaches 1-5k, after which point it levels off. The fanout service converges faster than the other two because the depth of the RPC dependency graph and the CBN are much shallower. Since training time grows linearly with the training size, there is no benefit from collecting a larger training dataset after the model's accuracy converges.

6.3.2 Tracing Frequency. We also explored the impact of tracing frequency of detection accuracy. Figure 16 shows the detection accuracy of Sage as the sampling frequency changes for the Chain service, the Fanout service, and the Social Network; the results are similar for the other services. The training dataset is collected over

24 hours, and we vary the sampling interval from one second to one minute. Since we are focused on non-transient faults, whose underlying causes cannot resolve themselves for an extended period of time without external intervention, the sampling frequency does not affect the observability of the error. QoS for individual microservices typically ranges from hundreds of microseconds to a few milliseconds. A mechanism that relies on temporal patterns requires a microsecond-level sampling interval to discover causality, which is impractical in large-scale deployments [49]. On the contrary, Sage's detection accuracy does not change much as the sampling interval increases because it does not leverage temporal patterns in timeseries to detect root causes. As the sampling interval decreases, the detection accuracy increases slightly because higher sampling frequency helps mitigate overfitting.

6.4 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 from the end-to-end service.

The six updates are indicated by red dash lines labeled with A-F in Figure 17. 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` service closer to the back-end, between `url-shorten` and `url-shorten-mongodb`. The further downstream a new service is, the more neurons 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 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* uses all techniques of Sec. 4.4, which reuse the existing network parameters and only retrain the neurons that are impacted by the updates. All approaches are trained in parallel; a new data batch arrives every 30s.

6.4.1 Retraining Time. Retraining for *partial+incremental retraining* takes a few seconds and up to a few minutes for the largest data batches. Moreover, it 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.

6.4.2 Root Cause Detection Accuracy. Fig. 17 shows that *partial+incremental retraining* and *incremental retraining* have the lowest

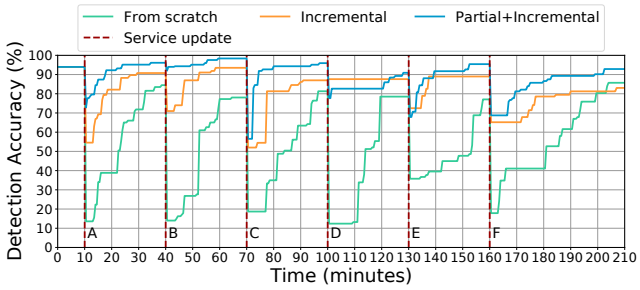

Figure 17: Detection accuracy for Sage, without and with partial & incremental retraining. Dash lines show when application updates are rolled out for the Social Network.

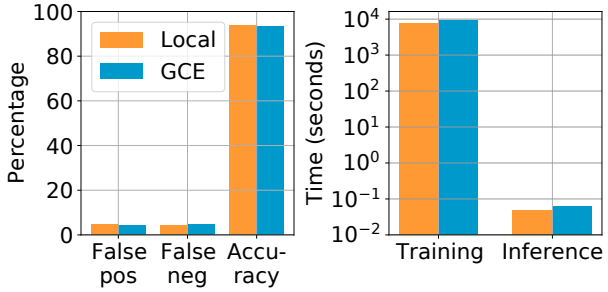

Figure 18: Sage's accuracy and speed on the local cluster and GCE.

accuracy drop immediately after an update. On the contrary, *retraining from scratch* almost 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.

6.5 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, depending on their resource needs, and shard the caches and databases. We simulate a graph of 1000 users.

We first validate Sage's accuracy compared to the local cluster. Fig. 18a shows that the accuracy on GCE is unchanged, indicating that Sage's ability to detect root causes is not impacted by system scale. Fig. 18b compares the training and inference time on the two clusters.

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. Although we deploy 6.7× more

containers on GCE, the training and inference times only increase by 19.4% and 26.5% respectively. In comparison, a similar increase in cluster size, resulted in an almost 4× increase in inference time for Seer [49]. 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 due to avoiding high-frequency, detailed tracing for root cause detection.

7 DISCUSSION

7.1 Cycles in RPC Dependencies

Generally, microservice graphs are DAGs, since cycles between tiers create positive feedback loops, which introduce failures and undermine the design principles of the microservices model. However, bidirectional streaming RPCs exist between two microservices, where the client and server both send a message sequence independently within a single request [5]. This cycle cannot be modeled by the CBN. To eliminate such cyclic dependencies, we merge both sides of the bidirectional streaming RPC into a metanode with both the client- and server-side latency, which shares the incoming and outgoing edges of both directions. The GVAE treats the metanode as a normal microservice.

7.2 Collecting Training Data

Sage leverages an unsupervised GVAE model that does not require data labeling. Therefore, it directly uses the tracing data collected in-situ by a cloud's monitoring infrastructure for training. As with any ML model, the quality of training data impacts accuracy. A primary challenge of cloud performance analysis is handling load variation [22]. Here variation is welcome, as it exposes a more diverse range of behaviors Sage can learn from. Nevertheless, it is still possible that a well-maintained system with few to no QoS violations has insufficient failure modes to train the model. In this case, Sage can leverage data obtained through fault injection tests with chaos engineering tools, such as Chaos Monkey [26], which are already in place in many cloud providers, including Netflix, Google, and Microsoft [6, 15, 26, 92].

7.3 Comparison with Seer, CauseInfer, and Microscope

Seer [49] is hybrid CNN+LSTM model used to predict performance issues in the near future and proactively prevent them. Compared to Seer, Sage leverages unsupervised learning which does not require labeling traces in the training set with the sources of QoS violations. This makes Sage easier to deploy in large-scale cloud environments, where injecting contentious benchmarks to initiate QoS violations is challenging. Additionally, Sage depends on lightweight tracing, and it does not require application- or kernel-level tracing to collect the number of outstanding requests across the system stack. Unlike Seer, Sage is a reactive tool, so even though it cannot avoid QoS violations altogether, it detects performance issues quickly, and applies corrective action before the QoS violation amplifies across dependent tiers.

CauseInfer [28] and Microscope [71] are two similar systems for performance diagnosis in distributed environments. They both use conditional cross entropy for conditional independence tests and the PC algorithm to build causal relationship DAGs between services. However, conditional independence is a difficult hypothesis to test for because conditional independence tests can suffer from type I error due to finite sample sizes, as shown in [97]. In addition, the worst-case complexity of the PC algorithm is exponential with the number of nodes in the graph, which limits the scalability of CauseInfer and Microscope. Sage outperforms CauseInfer and Microscope in terms of accuracy and scalability since it builds a non-strict causal DAG directly from the RPC dependency graph, and uses counterfactual queries to validate the causality for every event.

7.4 Limitations

Sage, as well as other data-driven methods, cannot detect the source of a performance issue if it has never observed a similar situation in the past. Through the latent variables in the model, Sage locates the problematic job associated with the root cause and flags it as the issue. Sage primarily focuses on deployment, configuration, and resource-related performance issues, since they directly correlate with the corresponding performance metrics. A similar methodology, with some additional application instrumentation, could be applied to also diagnose design bugs that initiate performance issues. We leave the root cause analysis of such non resource-related QoS violations to future work. In the current system, if the source of the QoS violation is not resource-related, i.e., all resource-related sources have been eliminated via counterfactuals, developers would need to be involved to examine if there is a software bug causing the QoS violation.

8 CONCLUSIONS

We have presented Sage, an ML-driven root cause analysis system for interactive cloud microservices. Unlike prior work, Sage leverages entirely unsupervised ML models to detect the source of unpredictable performance, removing the need for empirical diagnosis or data labeling. Sage works online to detect and correct performance issues, while also adapting to changes in application design. In both small- and large-scale experiments, Sage achieves high accuracy in pinpointing the root cause of QoS violations. Given the increasing complexity of cloud services, automated, data-driven systems like Sage improve performance without sacrificing resource efficiency.

ACKNOWLEDGMENTS

We sincerely thank Landon Cox for his valuable feedback while shepherding our paper. We also sincerely thank Partha Ranganathan, Yi Ding, Yanqi Zhang, Neeraj Kulkarni, Shuang Chen, Yi Jiang, Nikita Lazarev, Zhuangzhuang Zhou, Liqun Cheng, Rama Govindaraju, and the anonymous reviewers for their feedback on earlier versions of this manuscript. This work was in part supported by an NSF CAREER Award CCF-1846046, NSF grant NeTS CSR-1704742, a Sloan Research Fellowship, a Microsoft Research Fellowship, an Intel Faculty Rising Star Award, a Facebook Research Faculty Award, and a John and Norma Balen Sesquisentennial Faculty Fellowship.

REFERENCES

[1] "Apache thrift," https://thrift.apache.org.

[2] "Decomposing twitter: Adventures in service-oriented architecture," https://www.slideshare.net/InfoQ/decomposing-twitter-adventures-in-serviceoriented-architecture.
[3] "giltene/wrk2," https://github.com/giltene/wrk2.
[4] "google/cadvisor," https://github.com/google/cadvisor.
[5] "grpc: A high performance open-source universal rpc framework," https://grpc.io/.
[6] "Inside azure search: Chaos engineering," https://azure.microsoft.com/en-us/blog/inside-azure-search-chaos-engineering/.
[7] "Jaeger: open source, end-to-end distributed tracing," https://www.jaegertracing.io/.
[8] "Opentracing," https://opentracing.io/.
[9] "Prometheus," https://prometheus.io/.
[10] "prometheus/blackbox_exporter," https://github.com/prometheus/blackbox_exporter.
[11] "prometheus/node_exporter," https://github.com/prometheus/node_exporter.
[12] "Spring framework," https://spring.io/projects/spring-framework.
[13] "stress-ng," https://wiki.ubuntu.com/Kernel/Reference/stress-ng.
[14] "tc-netem(8) - linux manual page," http://man7.org/linux/man-pages/man8/tc-netem.8.html.
[15] "What facebook has learned from regularly shutting down entire data centers," https://www.datacenterknowledge.com/archives/2016/08/31/facebook-learned-regularly-shutting-entire-data-centers.
[16] "Zipkin," http://zipkin.io.
[17] "The evolution of microservices," https://www.slideshare.net/adriancockcroft/evolution-of-microservices-craft-conference, 2016.
[18] "Microservices workshop: Why, what, and how to get there," http://www.slideshare.net/adriancockcroft/microservices-workshop-craft-conference.
[19] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 74–89. [Online]. Available: https://doi.org/10.1145/945445.945454
[20] "Amazon ec2," http://aws.amazon.com/ec2/.
[21] B. D. Anderson, M. Deistler, and J.-M. Dufour, "On the sensitivity of granger causality to errors-in-variables, linear transformations and subsampling," *Journal of Time Series Analysis*, vol. 40, no. 1, pp. 102–123, 2019.
[22] D. Ardelean, A. Diwan, and C. Erdman, "Performance analysis of cloud applications," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 405–417. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/ardelean
[23] M. Attariyan, M. Chow, and J. Flinn, "X-ray: Automating root-cause diagnosis of performance anomalies in production software," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX, 2012, pp. 307–320.
[24] M. Azure, *Azure Monitor documentation*, 2020. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-monitor/
[25] L. Barroso and U. Hoelzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. MC Publishers, 2009.
[26] A. Basiri, N. Behnam, R. De Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.
[27] L. M. d. Campos, "A scoring function for learning bayesian networks based on mutual information and conditional independence tests," *Journal of Machine Learning Research*, vol. 7, no. Oct, pp. 2149–2187, 2006.
[28] 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.
[29] S. Chen, C. Delimitrou, and J. F. Martinez, "PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services," in *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2019.
[30] X. Chen, M. Zhang, M. Mao, and P. Bahl, "Automating network application dependency discovery: Experiences, limitations, and new solutions," in *Proc. of OSDI*. 2008.
[31] Z. Chen and B. Liu, "Lifelong machine learning," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 12, no. 3, pp. 1–207, 2018.
[32] L. Cherkasova, K. Ozonat, Ningfang Mi, J. Symons, and E. Smirni, "Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, 2008, pp. 452–461.
[33] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch, "The mystery machine: End-to-end performance analysis of large-scale internet services," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 217–231.
[34] G. Cloud, *Cloud Monitoring documentation*, 2020. [Online]. Available: https://cloud.google.com/monitoring/docs/apis
[35] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase, "Correlating instrumentation data to system states:a building block for automated diagnosis and control," in *HP Laboratories Palo Alto, HPL-2004-183, October 19, 2004.*
[36] 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*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 153–167.
[37] J. Dean and L. A. Barroso, "The tail at scale," in *CACM, Vol. 56 No. 2.*
[38] 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.
[39] ——, "QoS-Aware Scheduling in Heterogeneous Datacenters with Paragon," in *ACM Transactions on Computer Systems (TOCS), Vol. 31 Issue 4*. December 2013.
[40] ——, "Quality-of-Service-Aware Scheduling in Heterogeneous Datacenters with Paragon," in *IEEE Micro Special Issue on Top Picks from the Computer Architecture Conferences*. May/June 2014.
[41] ——, "Quasar: Resource-Efficient and QoS-Aware Cluster Management," in *Proc. of ASPLOS*. Salt Lake City, 2014.
[42] ——, "HCloud: Resource-Efficient Provisioning in Shared Cloud Systems," in *Proceedings of the Twenty First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2016.
[43] ——, "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.
[44] 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.
[45] F. C. Eigler, V. Prasad, W. Cohen, H. Nguyen, M. Hunt, J. Keniston, and B. Chen, "Architecture of systemtap: a linux trace/probe tool," 2005.
[46] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: A pervasive network tracing framework," in *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, ser. NSDI'07. Berkeley, CA, USA: USENIX Association, 2007, pp. 20–20.
[47] Y. Gan, M. Pancholi, D. Cheng, S. Hu, Y. He, and C. Delimitrou, "Seer: Leveraging Big Data to Navigate the Complexity of Cloud Debugging," in *Proceedings of the Tenth USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, July 2018.
[48] 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.
[49] 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.
[50] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, "Adaptive, model-driven autoscaling for cloud applications," in *11th International Conference on Autonomic Computing (ICAC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 57–64. [Online]. Available: https://www.usenix.org/conference/icac14/technical-sessions/presentation/gandhi
[51] M. L. Ginsberg, "Counterfactuals," *Artificial intelligence*, vol. 30, no. 1, pp. 35–79, 1986.
[52] "Google container engine," https://cloud.google.com/container-engine.
[53] R. M. Gray, *Entropy and information theory*. Springer Science & Business Media, 2011.
[54] M. Grechanik, C. Fu, and Q. Xie, "Automatically finding performance problems with feedback-directed learning software testing," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 156–166.
[55] J. Hauswald, M. A. Laurenzano, Y. Zhang, and et al., "Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers," in *Proc. of ASPLOS*, 2015.
[56] I. Higgins, L. Matthey, A. Pal, C. Burgess, X. Glorot, M. Botvinick, S. Mohamed, and A. Lerchner, "beta-vae: Learning basic visual concepts with a constrained variational framework," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. [Online]. Available: https://openreview.net/forum?id=Sy2fzU9gl
[57] C. Hitchcock, "Probabilistic causation," in *The Stanford Encyclopedia of Philosophy*, fall 2018 ed., E. N. Zalta, Ed. Metaphysics Research Lab, Stanford University, 2018.

[58] M. Höfler, "Causal inference based on counterfactuals," *BMC medical research methodology*, vol. 5, no. 1, p. 28, 2005.

[59] S. C. Hoi, D. Sahoo, J. Lu, and P. Zhao, "Online learning: A comprehensive survey," *arXiv preprint arXiv:1802.02871*, 2018.

[60] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, "Performance anomaly detection and bottleneck identification," *ACM Comput. Surv.*, vol. 48, no. 1, Jul. 2015.

[61] O. Ivanov, M. Figurnov, and D. Vetrov, "Variational autoencoder with arbitrary conditioning," 2018.

[62] V. Jeyakumar, O. Madani, A. Parandeh, A. Kulshreshtha, W. Zeng, and N. Yadav, "Explainit! – a declarative root-cause analysis engine for time series data," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 333–348. [Online]. Available: https://doi.org/10.1145/3299869.3314048

[63] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 77–88.

[64] G. B. Jr. and R. Koenker, "An empirical quantile function for linear models with iid errors," *Journal of the American Statistical Association*, vol. 77, no. 378, pp. 407–415, 1982. [Online]. Available: https://doi.org/10.1080/01621459.1982.10477826

[65] M. Kalisch and P. Bühlmann, "Estimating high-dimensional directed acyclic graphs with the pc-algorithm," *Journal of Machine Learning Research*, vol. 8, no. Mar, pp. 613–636, 2007.

[66] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang, "Grandslam: Guaranteeing slas for jobs in microservices execution frameworks," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3302424.3303958

[67] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *arXiv preprint arXiv:1312.6114*, 2013.

[68] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska *et al.*, "Overcoming catastrophic forgetting in neural networks," *Proceedings of the national academy of sciences*, vol. 114, no. 13, pp. 3521–3526, 2017.

[69] D. Koller and N. Friedman, *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[70] C.-C. Lin, P. Liu, and J.-J. Wu, "Energy-aware virtual machine dynamic provision and scheduling for cloud computing," in *Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing (CLOUD)*. Washington, DC, USA, 2011. [Online]. Available: http://dx.doi.org/10.1109/CLOUD.2011.94

[71] 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.

[72] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," in *Proceedings of the 41st Annual International Symposium on Computer Architecuture (ISCA)*. Minneapolis, MN, 2014.

[73] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proc. of the 42Nd Annual International Symposium on Computer Architecture (ISCA)*. Portland, OR, 2015.

[74] C. Louizos, U. Shalit, J. M. Mooij, D. Sontag, R. Zemel, and M. Welling, "Causal effect inference with deep latent-variable models," in *Advances in Neural Information Processing Systems*, 2017, pp. 6446–6456.

[75] J. Mars and L. Tang, "Whare-map: heterogeneity in "homogeneous" warehouse-scale computers," in *Proceedings of ISCA*. Tel-Aviv, Israel, 2013.

[76] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proceedings of MICRO*. Porto Alegre, Brazil, 2011.

[77] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, "Power management of online data-intensive services," in *Proceedings of the 38th annual international symposium on Computer architecture*, 2011, pp. 319–330.

[78] P. Menzies, "Counterfactual theories of causation," *Stanford Encyclopedia of Philosophy*, 2008.

[79] M. Moore, "Causation in the law," in *The Stanford Encyclopedia of Philosophy*, winter 2019 ed., E. N. Zalta, Ed. Metaphysics Research Lab, Stanford University, 2019.

[80] S. L. Morgan and C. Winship, *Counterfactuals and causal inference*. Cambridge University Press, 2015.

[81] K. Nagaraj, C. Killian, and J. Neville, "Structured comparative analysis of systems logs to diagnose performance problems," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 353–366.

[82] R. Nathuji, C. Isci, and E. Gorbatov, "Exploiting platform heterogeneity for power efficient data centers," in *Proceedings of ICAC*. Jacksonville, FL, 2007.

[83] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: Managing performance interference effects for qos-aware clouds," in *Proceedings of EuroSys*. Paris,France, 2010.

[84] R. E. Neapolitan *et al.*, *Learning bayesian networks*. Pearson Prentice Hall Upper Saddle River, NJ, 2004, vol. 38.

[85] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, May 2015, pp. 293–307.

[86] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," in *Proceedings of SOSP*. Farminton, PA, 2013.

[87] G. I. Parisi, R. Kemker, J. L. Part, C. Kanan, and S. Wermter, "Continual lifelong learning with neural networks: A review," *Neural Networks*, 2019.

[88] J. Pearl *et al.*, "Causal inference in statistics: An overview," *Statistics surveys*, vol. 3, pp. 96–146, 2009.

[89] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan, "Gorilla: A fast, scalable, in-memory time series database," *Proc. VLDB Endow.*, vol. 8, no. 12, p. 1816–1827, Aug. 2015. [Online]. Available: https://doi.org/10.14778/2824032.2824078

[90] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt, "Google-wide profiling: A continuous profiling infrastructure for data centers," *IEEE Micro*, pp. 65–79, 2010. [Online]. Available: http://www.computer.org/portal/web/csdl/doi/10.1109/MM.2010.68

[91] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat, "Wap5: Black-box performance debugging for wide-area systems," in *Proceedings of the 15th International Conference on World Wide Web*, ser. WWW '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 347–356.

[92] J. Robbins, K. Krishnan, J. Allspaw, and T. A. Limoncelli, "Resilience engineering: learning to embrace failure," *Queue*, vol. 10, no. 9, pp. 20–28, 2012.

[93] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015. [Online]. Available: http://networkrepository.com

[94] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, and J. Wilkes, "Autopilot: Workload autoscaling at google," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3387524

[95] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *Proceedings of EuroSys*. Prague, 2013.

[96] A. W. Services, *Amazon CloudWatch User Guide Document History*, 2020. [Online]. Available: https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring

[97] R. D. Shah and J. Peters, "The hardness of conditional independence testing and the generalised covariance measure," *Ann. Statist.*, vol. 48, no. 3, pp. 1514–1538, 06 2020. [Online]. Available: https://doi.org/10.1214/19-AOS1857

[98] H. Shan, Y. Chen, H. Liu, Y. Zhang, X. Xiao, X. He, M. Li, and W. Ding, "??-diagnosis: Unsupervised and real-time diagnosis of small- window long-tail latency in large-scale microservice platforms," in *The World Wide Web Conference*, ser. WWW '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 3215–3222. [Online]. Available: https://doi.org/10.1145/3308558.3313653

[99] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: elastic resource scaling for multi-tenant cloud systems," in *Proceedings of SOCC*. Cascais, Portugal, 2011.

[100] 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. [Online]. Available: https://research.google.com/archive/papers/dapper-2010-1.pdf

[101] A. Silvestrini and D. Veredas, "Temporal aggregation of univariate and multivariate time series models: a survey," *Journal of Economic Surveys*, vol. 22, no. 3, pp. 458–497, 2008.

[102] M. Slee, A. Agarwal, and M. Kwiatkowski, "Thrift: Scalable cross-language services implementation," *Facebook White Paper*, vol. 5, no. 8, 2007.

[103] K. Sohn, H. Lee, and X. Yan, "Learning structured output representation using deep conditional generative models," in *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds. Curran Associates, Inc., 2015, pp. 3483–3491.

[104] A. Sriraman and T. F. Wenisch, "μ suite: A benchmark suite for microservices," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, 2018, pp. 1–12.

[105] A. Sriraman and A. Dhanotia, "Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 733–750. [Online]. Available: https://doi.org/10.1145/3373376.3378450

[106] A. Sriraman, A. Dhanotia, and T. F. Wenisch, "Softsku: Optimizing server architectures for microservice diversity @scale," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 513–526. [Online]. Available: https://doi.org/10.1145/3307650.3322227

[107] A. Sriraman and T. F. Wenisch, "µtune: Auto-tuned threading for OLDI microservices," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 177–194. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/sriraman

[108] L. Suresh, P. Bodik, I. Menache, M. Canini, and F. Ciucu, "Distributed resource management across process boundaries," in *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*. Santa Clara, CA, 2017.

[109] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan, "Prepare: Predictive performance anomaly prevention for virtualized cloud systems," in *Proc. of the 32nd IEEE International Conference on Distributed Computing Systems*. 2012.

[110] J. Teoh, M. A. Gulzar, G. H. Xu, and M. Kim, "Perfdebug: Performance debugging of computation skew in dataflow systems," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 465–476.

[111] J. Thalheim, A. Rodrigues, I. E. Akkus, P. Bhatotia, R. Chen, B. Viswanath, L. Jiao, and C. Fetzer, "Sieve: Actionable insights from monitored metrics in distributed systems," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, ser. Middleware '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 14–27. [Online]. Available: https://doi.org/10.1145/3135974.3135977

[112] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[113] C. Wang, K. Viswanathan, L. Choudur, V. Talwar, W. Satterfield, and K. Schwan, "Statistical techniques for online anomaly detection in data centers," in *12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops*, 2011, pp. 385–392.

[114] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang, "Automatic misconfiguration troubleshooting with peerpressure," in *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, ser. OSDI'04. USA: USENIX Association, 2004, p. 17.

[115] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: precise online qos management for increased utilization in warehouse scale computers," in *Proceedings of ISCA*. 2013.

[116] J. Yoon, E. Yang, J. Lee, and S. J. Hwang, "Lifelong learning with dynamically expandable networks," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=Sk7KsfW0-