Dynamic Load Scaling via Distributed Trace Sampling & Analysis

Kevin Mooney, John Durkin, Samson Koshy, Sushil Khadka, Reza Farivar, Jiawei Tyler Gu

University of Illinois at Urbana-Champaign, Urbana, IL, USA
{kmoone3, jdurkin3, skosh3, sushil2, farivar2, jiaweig3}@illinois.edu

***Abstract*—Modern systems often use automation for resource scaling. Current automatic scaling systems focus on system-generated metrics like CPU and memory utilization to determine scaling. While these indicators are precise, they often lack insights into the operational performance of applications affected by scaling. Distributed traces model end-to-end requests or transactions as a hierarchical graph of spans, each representing a measured execution point. Trace data can reveal insights into latencies that are not captured by system-level metrics. To address this, we introduce SCALE, an automated resource scaler that uses distributed traces for scaling decisions. SCALE samples trace spans for duration anomalies and processes them through a heuristics-based modeling engine against defined latency thresholds. If thresholds are exceeded, SCALE triggers scaling commands for orchestration systems. SCALE is evaluated using trace data from open-source microservice benchmarks in a Kubernetes cluster.**

I Introduction

Whether running workloads on remote cloud platforms, on-premises clusters, or compute grids, proper resource utilization is one of the most significant factors that can affect operational expenses and capital expenditures. On one end of the spectrum, under-utilizing resources results in wasted costs through unused hardware and network allocation. Conversely, over-utilizing resources can negatively impact response times, causing degradation and failures throughout your application. At a minimum, this can frustrate the client base, while in severe cases, it can lead to detrimental production outages, potentially causing irreparable damage to the company or brand.

Existing technologies for dynamic load scaling are limited. [1] focuses directly on hardware resources such as CPU and memory, where new instances will be created when CPU utilization exceeds a predefined threshold. Some systems delve deeper, relying on application-generated metrics. However, this involves keeping track of the right parts of your application and manually connect the appropriate application metrics to the right scaling actions.

We propose SCALE, a novel technique utilizing distributed tracing data to make scaling decisions based on application performance, as opposed to solely relying on simple metrics. SCALE does not depend on a specific decision-making mechanism or orchestration technology; instead, it provides abstraction around these two general areas and focuses on differentiating interesting from benign trace data. Such a design would allow research to concentrate on anomaly detection within trace data, enabling operators to customize scaling for diverse and heterogeneous architectures. SCALE emphasizes on the sampling of distributed tracing for anomalous execution durations. In cases where anomalies are detected, the SCALE processor will generate scaling signals based on the analysis of tail latencies of individual calls within those graphs.

II Background & Characterization

A. Related Works

In our research, we drew particular motivation from gaps in the current state of the art of autoscaling systems pointed out by our fellow researchers. Much of this fell in line with our vision, and helped reinforce the direction we took in our own implementation.

1) Autoscaling: Straesser et al. [1] argue that state-of-the-art autoscalers are often too complex for production and rely heavily on CPU metrics. However, performance for some applications is not driven by CPU or memory, making a combination of platform and application metrics beneficial. However, there is also a lack of general-purpose autoscalers for these needs.

Eder et al. [2] emphasize the importance of considering deployment costs in pay-per-use models and ensuring distributed tracing does not degrade application performance. They found that the performance impact—measured in runtime, memory usage, and initialization—varies with tools like Zipkin [3], Otel [4], and SkyWalking [5]. They conclude that with proper tool selection, the benefits of distributed tracing can outweigh performance downsides.

Yu et al. [6] proposed MicroScaler, which identifies services needing scaling to meet SLA requirements by collecting service metrics (QoS, latency) through proxy sidecars in microservice architectures. While novel, it bases scaling solely on latency and introduces sidecar overheads.

2) Sampling: Modern distributed applications are complex, needing deep insights into request orchestration across services. Distributed tracing provides this view but can generate terabytes of trace data daily, complicating pro-

cessing and storage. Sampling helps maintain efficiency while managing data overload.

The common strategy used by tools like Jaeger [7] and Zipkin [3] is uniform random sampling, or head-based sampling, where decisions are made at a trace's start. This method often captures redundant traces, limiting anomaly detection value.

To improve this, tail-based sampling defers decisions until a trace is complete, focusing on traces likely to be informative or anomalous. It is widely adopted in academia and industry. For instance, Tracemesh [8] uses DenStream [9] for clustering streaming data, sampling traces based on their evolving characteristics to cut back on oversampling. Sieve [10] applies a biased sampling method using attention scores with Robust Random Cut Forests (RRCF) to identify uncommon traces. Both leverage trace structural and temporal variations to boost sampling effectiveness. Sifter [11] prioritizes diverse traces by weighting sampling decisions towards those that are underrepresented in its model, thus improving trace variety. Las-Casas et al. [12] proposed a weighted sampling method using a hierarchical clustering technique called Purity Enhancing Rotations for Cluster Hierarchies (PERCH) to ensure diversity in selected traces.

B. Human Insight vs. Machine Learning

When surveying the landscape of current autoscaling systems, one significant choice that arises is whether to take a heuristics-based approach or to use a fully automated approach. Many original scaling systems were based on heuristics. It is, of course, easier to implement a simple rules-based parser than to design a fully automated system. This allows you to leverage common domain knowledge and directly apply scaling semantics that you believe best serve the health of your system. However, this approach has some obvious drawbacks. The most notable issue is the constant need for human intervention. You will undoubtedly spend well-focused man-hours devising your scaling rule set, and then whatever time it takes to put pen to paper. This is also not a one-time cost. As your system evolves due to planned architecture changes or unforeseen circumstances, you will continuously need to rework your scaling model. Beyond the manual overhead that this approach entails, there's the fact that humans are not infallible; any scheme they devise, while hopefully sound, is unlikely to be fully optimal.

III Design

A. Overview

We introduce SCALE, a software system which analyzes various system and application generated observability data in order to perform dynamic scaling of application work loads in cloud hosted and/or on-premises environments. We envision a monitor that can make use of open technologies to probe distributed tracing and metric data.

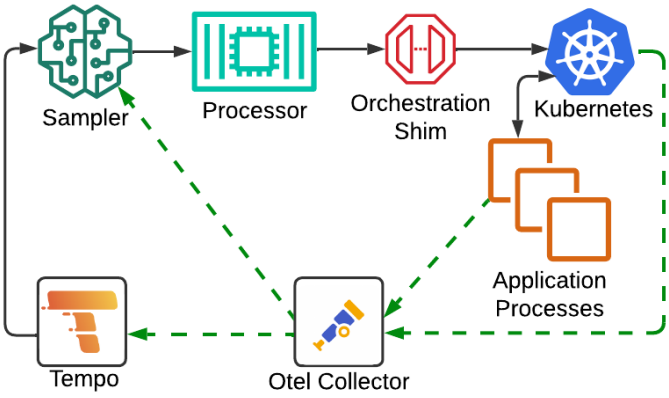

Fig. 1. SCALE Architecture

This system would wire up to client APIs of a diverse set of process orchestration software. This can include container orchestrators, serverless function controllers or VM provisioning frameworks amongst others. Based on processing of the observability data, a processor will make vertical or horizontal scaling decisions. These decisions will then be communicated to the orchestration systems via their client API to carry out the scaling action.

While we aim for extensibility, the true novelty of our research hinges on the viability of making scaling decisions based on distributed tracing data. Hence, our implementation will directly focus on the consumption, sampling and analysis of traces. The analysis centers around a heuristics based trace processor which will check trace spans for latency breaches above configured thresholds. Based on actions attached to those thresholds, when breached, the processor will initiate scaling directives via a shim that sits atop of the Kubernetes API.

Figure 1 represents the core architecture of SCALE. The three major components which make up SCALE itself are the *sampler*, *processor*, and *orchestration shim*. These components work together to form a cohesive, modular workload scaler.
SCALE is dependent upon a distributed trace storage mechanism and a distributed trace delivery mechanism which leverage OpenTelemetry [4]. In our implementation we use Grafana Tempo [13] and OpenTelemetry Collector respectively.

B. Sampler

The sampler is the first direct SCALE component within the architecture's overall pipeline. The job of the sampler is to sample trace spans for anomalies before they are sent downstream for scaling determination. This allows SCALE to predetermine interesting traces in a fast manner, so that extra resources are not wasted on a deep analysis of the call graph. Downstream the SCALE processor relies on a heuristics based examination of traces. However, there is a non-insignificant overhead involved in analyzing large call graphs that should be avoided if possible. This would be

further exacerbated if the processor (described in Section III-B4) were swapped for an implementation that relied on a more compute intensive approach.

At it's core the main purpose of the sampler is to identify trace spans that have unusual durations. Many of the researched works attempt to perform a holistic sampling of distributed traces. This is often to identify anomalies in both newly seen call graphs in addition to call durations. In our case we are only interested in anomalies in the duration of calls. Given this our sampling is performed directly on each span individually. This leads to both a more deterministic and more performant implementation.

Spans in open telemetry contain four distinct attributes that are important to our sampling. The first of these two are the service name and operation name. The service name is taken to mean a collection of instances of a specific service as opposed to a single running instance. The operation name denotes a specific unit of execution within a service. Together these help us to uniquely identify very specific call points within a distributed system for classification. The second of the two attributes are start and end time in nanoseconds since Unix epoch. Each span S is thus characterized by:

- **Service name:** s
- **Operation name:** o
- **Start time:** t_{start}
- **End time:** t_{end}

The **Duration** D of a span is computed as:

$$D = t_{\text{end}} - t_{\text{start}} \tag{1}$$

The sampler uses Half-Space-Trees (HS-Tree) [14] to perform anomaly detection on the durations of spans. An online variant of isolation forests, HS-Tree provides a model for fast one-class anomaly detection within evolving data streams. With HS-Tree you provide bounds on the data space which features can fall within. Multiple trees are then formed based on whether features fall within one half of the bound or the other, with different trees holding different orderings of features. All trees are sampled, and a prediction is formed from a consensus of the results from all trees. HS-Tree is best suited for cases where anomalies are rare which is the expectation in distributed trace data. The HS-Tree model isolates anomalies based on isolation depth h, producing an anomaly score:

$$\text{Anomaly Score} = 2^{\frac{-h}{H}} \tag{2}$$

where: $\quad \text{H} = \text{maximum tree height}$

A number of proven open source implementations exist, and for SCALE we chose to go with from River [15] an online machine learning library for Python.

To categorize the service and operation name we use one-hot encoding. The river HS-Tree implementation is able to accept a sparse python dictionary that does not contain the entire feature set of your data space. Given this, we simply use simple numerically increasing integers as keys for the feature name of the service and operation.

Each span is represented by the following feature set (where *OneHotEncode* is defined in **Algorithm 1**):

$$\mathbf{X} = \{\text{OneHotEncode}(s, o), D\} \tag{3}$$

Algorithm 1 One-Hot Encoding Function

```
Initialize an empty dictionary encodings
Function OneHotEncode(service, operation)
call_name ← service + " :: " + operation
encoding ← encodings.get(call_name)
if encoding is None then
    encoding ← len(encodings)
    encodings[service] ← encoding
end if
record ← {str(encoding) : 1}
return record
End Function
```

We then take the difference of the end timestamp minus start timestamp and scale these to milliseconds. This is added as a second feature to the record and passed into the River HS-Tree model for sampling. To initially train the HS-Tree model, a pre-configured set of traces are queried from Tempo via an HTTP REST service which provides traces in OpenTelemetry protocol buffer format. The traces are stored in a pandas data frame as they are collected. When all traces have been pulled, the model is trained which each span featurized as per the above described algorithm. The sampler has six core parameters which are used to calibrate the model prior to training:

TABLE I
SAMPLER PARAMETERS

Parameter	Description
n_tree	number of HS-Trees
height	height of each HS-Tree
window_size	observations in each HS-Tree node
min_score	minimum score required for sampling
max_duration	maximum bound for duration feature
train_size	number of spans to train the model

After the model has been successfully trained according to the parameters, the sampler starts a gRPC listener which exposes two RPCs. The first is an implementation of the OTLP gRPC interface for receiving traces. As shown in figure 1, this allows Collector to stream new traces into the sampler as they are received from upstream applications and infrastructure. The second RPC provides a streaming endpoint which clients can connect to in order to receive sampled spans, which are provided to the client in their entirety as they were received from Collector. These together along with the core sampler logic essentially form a streaming pipeline of spans, with the OTLP endpoint as the source, the sampler logic as a filter and the client endpoint as a sink.

C. Processor

As described, the processor serves as a downstream component of the sampler within the SCALE architecture. It is the task of the processor to analyze traces and make the ultimate decision on whether to perform any scaling operations. The processor is configured with a set of rules defined via YAML configuration. These rules associate latency thresholds with scaling actions. An example of this is of the form *scale service A if operation B reaches a latency of X, N times within a given window.* The rules then drive the heuristics based analysis of the distributed traces received from the sampler.

Scaling actions are triggered when specific thresholds are exceeded:

$$\text{Action} \iff \text{Count}(D > T, \text{window}) \geq N \tag{4}$$

The system distinguishes between load bottlenecks and resource constraints, applying appropriate scaling decisions:

$$\text{Scale}(s, o) = \begin{cases} \text{Horizontal}, & \text{if load bottleneck} \\ \text{Vertical}, & \text{if resource bound} \end{cases} \tag{5}$$

The processor is composed of two main components, the consumer and processor engine. The consumer will initiate a streaming gRPC connection to receive the sampled spans using the Python asyncio version of gRPC. The spans received contain all of the same attributes and metadata that were also received by the sampler. These spans are somewhat hierarchical, but not organized according to traces, and more so to shared data between spans for compaction purposes. The spans are then dropped into an asyncio queue.

The processor engine will then read these spans from the queue. The spans are iterated over and flattened into a pandas dataframe. The rule set defined in the configuration file is then applied as precompiled predicates against the dataframe. Any rules whose criteria is a match has their actions folded into a conflated action set. The purpose of conflation is to deduplicate redundant scaling actions. When the analysis is complete the processor will then call the Kubernetes shim to perform the scaling action. In addition to scaling analysis and actions, the processor engine will also continually keep several statistics and trend analytics of inspected spans.

D. Orchestration Shim

The orchestration shim acts as a layer between the processor and Kubernetes itself. Its purpose is to abstract away the specific details of the Kubernetes API, in order to provide a more generic scaling interface to the processor. This abstraction model allows for the ability to swap out service orchestration backends, or more realistically to compose multiple backends. This is manifested in the form of an OrchestrationClient interface which provides a stub for scaling a resource, and another for getting the current scale of a resource. The shim is meant to be intentionally light weight, and implementations are called directly within the processor.

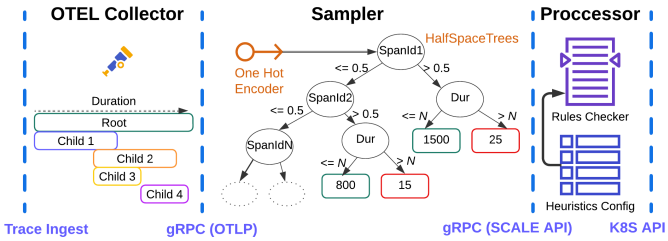

Fig. 2. SCALE Workflow

IV Evaluation

To evaluate our system, we will create a comprehensive testing environment that simulates real-world traffic patterns, bottlenecks, and resource constraints. This setup will allow for controlled comparisons between baseline autoscaling (CPU/memory-based) and SCALE. We will evaluate the autoscaling systems using open-source benchmarking suites, along with an in-house architecture resembling a personal finance application. To simulate production-like environments, we will use additional open-source simulation tools and chaos testing tools to generate realistic traffic surges and failure conditions across microservices. Our goal is to measure the speed and accuracy of autoscaling decisions made by the observability-driven system compared to the baseline.

A. Parameter Tuning

The performance of the proposed sampling mechanism was optimized by tuning critical parameters to achieve a balance across four key metrics:

1) False Positive Rate (FPR): FPR measures the proportion of normal spans that are incorrectly flagged as anomalous. This metric is critical for reducing noise in the system.

$$\text{FPR} = \frac{\text{False Positives (FP)}}{\text{Total Normal Spans}} \tag{6}$$

2) Accuracy: Accuracy quantifies the proportion of correctly classified spans, encompassing both true positives (TP) and true negatives (TN), relative to the total number of spans. Ensuring 100% accuracy was prioritized to avoid misclassification of both normal and anomalous spans.

$$\text{Accuracy} = \frac{\text{True Positives (TP)} + \text{True Negatives (TN)}}{\text{Total Spans}} \tag{7}$$

3) Processing Time Per Span: This metric reflects the computational efficiency of the sampler by averaging the time required to process each span. Reducing the value is critical to ensure scalability in high-throughput environments.

$$T_{span} = \frac{T_{total}}{TotalSpans} \tag{8}$$

4) F1 Score: The F1 Score provides a balanced measure of the sampler's precision and recall, capturing the trade-off between correctly identifying anomalies and minimizing false positives.

$$\text{F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{9}$$

where, $\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \quad \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$

The tuning process involved systematic adjustments of parameters such as the target anomaly score, feature scaling factors, height, number of trees, and window size. Each iteration was evaluated against the aforementioned metrics with the goal of minimizing FPR, maintaining 100% detection accuracy, reducing processing time, and maximizing the F1 score. These adjustments were guided by a feedback loop, enabling continuous improvement to optimize sampler performance.

B. Dataset Analysis

The optimized sampler was evaluated against two datasets used in previous studies *(GTrace [16] and TraceMesh [8])*. For both datasets, the sampler was benchmarked on the following metrics: precision, recall, F1 score, processing time, and FPR. Results demonstrated the sampler consistently maintained 100% accuracy, minimized FPR, and delivered competitive F1 scores while adhering to stringent efficiency requirements. Table II shows the results of these experiments, while 3 presents a confusion matrix visualizing accuracy against expected results.

TABLE II
DATASET COMPARISON

Metric	GTrace	TraceMesh
Total Spans	4881687	57908
Anomalies Detected	196987	4344
True Positives	81361	2895
False Positives	115626	1449
False Negatives	0	0
Precision	0.41302726	0.66643646
Recall	1.0	1.0
F1 Score	0.58459913	0.79983423
Processing Time	71µs	77µs

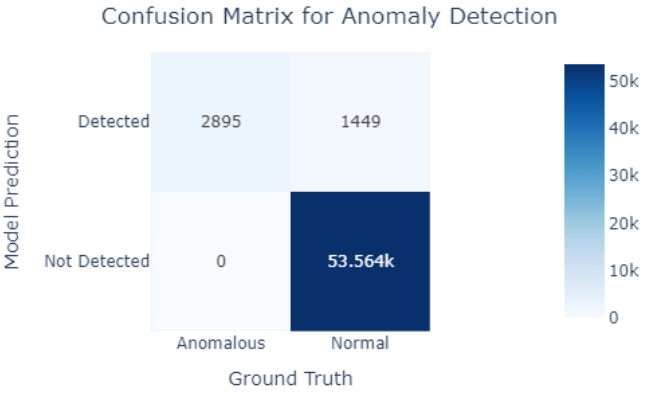

Fig. 3. TraceMesh Dataset Confusion Matrix

V CONCLUSION

In this paper we put forward the argument for a system capable of performing resource autoscaling decisions based upon distributed trace data as opposed to system level metrics, common in state of the art. To further this proposal, we implemented the SCALE framework. Rather than examine every trace that is generated by a distributed system, SCALE samples individual trace spans for anomalistic durations. Sampling is provided via half-space trees, a fast online anomaly detection model for evolving data streams. Sampled spans are then passed through a heuristics based processor that checks spans against a set of thresholds, and performs scaling actions if defined thresholds are breached. Through experimentation, we demonstrated performance and accuracy benefits of using half-space trees for span anomaly detection, as well as the benefit of utilizing sampled traces in general.

References

[1] M. Straesser, J. Grohmann, J. von Kistowski, S. Eismann, A. Bauer, and S. Kounev, "Why is it not solved yet? Challenges for production-ready autoscaling," in *Proceedings of the 2022 ACM/SPEC on international conference on performance engineering*, in ICPE '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 105–115. doi: 10.1145/3489525.3511680[1].

[2] C. Eder, S. Winzinger, and R. Lichtenthaler, "A comparison of distributed tracing tools in serverless applications," in *2023 IEEE international conference on service-oriented system engineering (SOSE)*, Los Alamitos, CA, USA: IEEE Computer Society, Jul. 2023, pp. 98–105. doi: 10.1109/SOSE58276.2023.00018[2].

[3] Zipkin, "OpenZipkin: A distributed tracing system." Accessed: Dec. 08, 2024. [Online]. Available: https://zipkin.io/

[4] OpenTelemetry, "OpenTelemetry." Accessed: Oct. 25, 2024. [Online]. Available: https://opentelemetry.io/

[5] Zipkin, "Apache SkyWalking." Accessed: Dec. 08, 2024. [Online]. Available: https://skywalking.apache.org/

[6] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Automatic scaling for microservices with an online learning approach," in *2019 IEEE international conference on web services (ICWS)*, 2019, pp. 68–75. doi: 10.1109/ICWS.2019.00023[3].

[7] Zipkin, "Jaeger: Open source, distributed tracing platform." Accessed: Dec. 08, 2024. [Online]. Available: https://www.jaegertracing.io/

[8] Z. Chen, Z. Jiang, Y. Su, M. R. Lyu, and Z. Zheng, "Tracemesh: Scalable and Streaming Sampling for Distribuauthor=Otero, Manuel and Garcia, Jose María and Fernandez, Pablo, booktitle=2024 IEEE 44th International Conference on Distributed Computing Systems Workshops (ICDCSW), title=Towards a lightweight distributed telemetry for microservices, year=2024, volume=, number=, pages=75-82, keywords=Root cause analysis;Biological system modeling;System performance;Ecosystems;Microservice architectures;Computer architecture;User interfaces;telemetry;oas;api;microservices, doi=10.1109/ICDCSW63686.2024.00018ted Traces ," in *2024 IEEE 17th international conference on cloud computing (CLOUD)*, Los Alamitos, CA, USA: IEEE Computer Society, Jul. 2024, pp. 54–65. doi: 10.1109/CLOUD62652.2024.00016[4].

[9] F. Cao, M. Estert, W. Qian, and A. Zhou, "Density-based clustering over an evolving data stream with noise," in *Proceedings of the 2006 SIAM international conference on data mining (SDM)*, 2006, pp. 328–339. doi: 10.1137/1.9781611972764.29[5].

[10] Z. Huang, P. Chen, G. Yu, H. Chen, and Z. Zheng, "Sieve: Attention-based sampling of end-to-end trace data in distributed microservice systems," in *2021 IEEE international conference on web services (ICWS)*, 2021, pp. 436–446. doi: 10.1109/ICWS53863.2021.00063[6].

[11] P. Las-Casas, G. Papakerashvili, V. Anand, and J. Mace, "Sifter: Scalable sampling for distributed traces, without feature engineering," in *Proceedings of the ACM symposium on cloud computing*, in SoCC '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 312–324. doi: 10.1145/3357223.3362736[7].

[12] P. Las-Casas, J. Mace, D. Guedes, and R. Fonseca, "Weighted sampling of execution traces: Capturing more needles and less hay," in *Proceedings of the ACM symposium on cloud computing*, in SoCC '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 326–332. doi: 10.1145/3267809.3267841[8].

[13] Grafana, "Grafana tempo OSS: Distributed tracing backend." Accessed: Oct. 25, 2024. [Online]. Available: https://grafana.com/oss/tempo/

[14] S. C. Tan, K. M. Ting, and T. F. Liu, "Fast anomaly detection for streaming data," in *Proceedings of the twenty-second international joint conference on artificial intelligence - volume volume two*, in IJCAI'11. Barcelona, Catalonia, Spain: AAAI Press, 2011, pp. 1511–1516.

[15] River, "River." Accessed: Nov. 15, 2024. [Online]. Available: https://riverml.xyz/dev/

[16] Z. Xie *et al.*, "From point-wise to group-wise: A fast and accurate microservice trace anomaly detection approach," in *Proceedings of the 31st ACM joint european software engineering conference and symposium on the foundations of software engineering*, in ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, pp. 1739–1749. doi: 10.1145/3611643.3613861[9].

[17] S. Luo *et al.*, "The power of prediction: Microservice auto scaling via workload learning," in *Proceedings of the 13th symposium on cloud computing*, in SoCC '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 355–369. doi: 10.1145/3542929.3563477[10].

[18] S. Luo *et al.*, "Characterizing microservice dependency and performance: Alibaba trace analysis," in *Proceedings of the ACM symposium on cloud computing*, in SoCC '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 412–426. doi: 10.1145/3472883.3487003[11].

[19] J. Shen *et al.*, "Network-centric distributed tracing with DeepFlow: Troubleshooting your microservices in zero code," in *Proceedings of the ACM SIGCOMM 2023 conference*, in ACM SIGCOMM '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 420–437. doi: 10.1145/3603269.3604823[12].

[20] D. Cha and Y. Kim, "Service mesh based distributed tracing system," in *2021 international conference on information and communication technology convergence (ICTC)*, 2021, pp. 1464–1466. doi: 10.1109/ICTC52510.2021.9620968[13].

[21] S. Ashok, V. Harsh, B. Godfrey, R. Mittal, S. Parthasarathy, and L. Shwartz, "TraceWeaver: Distributed request tracing for microservices without application modification," in *Proceedings of the ACM SIGCOMM 2024 conference*, in ACM SIGCOMM '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 828–842. doi: 10.1145/3651890.3672254[14].

[22] M. Toslali *et al.*, "An online probabilistic distributed tracing system." 2024. Available: https://arxiv.org/abs/2405.15645

[23] Y. Zhang, R. Isaacs, Y. Yue, J. Yang, L. Zhang, and Y. Vigfusson, "LatenSeer: Causal modeling of end-to-end latency distributions by harnessing distributed tracing," in *Proceedings of the 2023 ACM symposium on cloud computing*, in SoCC '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 502–519. doi: 10.1145/3620678.3624787[15].

[24] H. Huang *et al.*, "TraStrainer: Adaptive sampling for distributed traces with system runtime state," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024, doi: 10.1145/3643748[16].

[25] S. He *et al.*, "STEAM: Observability-preserving trace sampling," in *Proceedings of the 31st ACM joint european software engineering conference and symposium on the foundations of software engineering*, in ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, pp. 1750–1761. doi: 10.1145/3611643.3613881[17].

[26] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices," in *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, USENIX Association, Nov. 2020, pp. 805–825. Available: https://www.usenix.org/conference/osdi20/presentation/qiu

[27] V. Sachidananda and A. Sivaraman, "Erlang: Application-aware autoscaling for cloud microservices," in *Proceedings of the nineteenth european conference on computer systems*, in EuroSys '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 888–923. doi: 10.1145/3627703.3650084[18].

[28] G. Christofidi, K. Papaioannou, and T. D. Doudali, "Is machine learning necessary for cloud resource usage forecasting?" in *Proceedings of the 2023 ACM symposium on cloud computing*, in SoCC '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 544–554. doi: 10.1145/3620678.3624790[19].

[29] J. Liu, Q. Wang, S. Zhang, L. Hu, and D. Da Silva, "Sora: A latency sensitive approach for microservice soft resource adaptation," in *Proceedings of the 24th international middleware conference*, in Middleware '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 43–56. doi: 10.1145/3590140.3592851[20].

[30] H. Qiu *et al.*, "AWARE: Automate workload autoscaling with reinforcement learning in production cloud systems," in *2023 USENIX annual technical conference (USENIX ATC 23)*, Boston, MA: USENIX Association, Jul. 2023, pp. 387–402. Available: https://www.usenix.org/conference/atc23/presentation/qiu-haoran

[31] M. Fourati, S. Marzouk, and M. Jmaiel, "Towards microservices-aware autoscaling: A review," in *2023 IEEE symposium on computers and communications (ISCC)*, Los Alamitos, CA, USA: IEEE Computer Society, Jul. 2023, pp. 1080–1083. doi: 10.1109/ISCC58397.2023.10218213[21].

[32] T.-T. Nguyen, Y.-J. Yeom, T. Kim, D.-H. Park, and S. Kim, "Horizontal pod autoscaling in kubernetes for elastic container orchestration," *Sensors*, vol. 20, no. 16, 2020, doi: 10.3390/s20164621[22].

[33] E. Manzoor, S. M. Milajerdi, and L. Akoglu, "Fast memory-efficient anomaly detection in streaming heterogeneous graphs," in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, in KDD '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 1035–1044. doi: 10.1145/2939672.2939783[23].

[34] M. Otero, J. M. Garcia, and P. Fernandez, "Towards a lightweight distributed telemetry for microservices," in *2024 IEEE 44th international conference on distributed computing systems workshops (ICDCSW)*, 2024, pp. 75–82. doi: 10.1109/ICDCSW63686.2024.00018[24].

[35] G. Yu, P. Chen, Y. Li, H. Chen, X. Li, and Z. Zheng, "Nezha: Interpretable fine-grained root causes analysis for microservices on multi-modal observability data," in *Proceedings of the 31st ACM joint european software engineering conference and symposium on the foundations of software engineering*, in ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, pp. 553–565. doi: 10.1145/3611643.3616249[25].

[36] A. Belkhiri, M. Ben Attia, F. Gohring De Magalhaes, and G. Nicolescu, "Towards efficient diagnosis of performance bottlenecks in microservice-based applications (work in progress paper)," in *Companion of the 15th ACM/SPEC international conference on performance engineering*, in ICPE '24 companion. New York, NY, USA: Association for Computing Machinery, 2024, pp. 40–46. doi: 10.1145/3629527.3651432[26].

[37] A. Belkhiri, A. Shahnejat Bushehri, F. Gohring de Magalhaes, and G. Nicolescu, "Transparent trace annotation for performance debugging in microservice-oriented systems (work in progress paper)," in *Companion of the 2023 ACM/SPEC international conference on performance engineering*, in ICPE '23 companion. New York, NY, USA: Association for Computing Machinery, 2023, pp. 25–32. doi: 10.1145/3578245.3585030[27].

[38] L. Huang and T. Zhu, "Tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces," in *Proceedings of the ACM symposium on cloud computing*, in SoCC '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 76–91. doi: 10.1145/3472883.3486994[28].

[39] Minikube, "Minikube." Accessed: Oct. 26, 2024. [Online]. Available: https://minikube.sigs.k8s.io/

[40] E. Manzoor, S. M. Milajerdi, and L. Akoglu, "Fast memory-efficient anomaly detection in streaming heterogeneous graphs," in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, in KDD '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 1035–1044. doi: 10.1145/2939672.2939783[29].

[41] S. Bhatia, M. Wadhwa, K. Kawaguchi, N. Shah, P. S. Yu, and B. Hooi, "Sketch-based anomaly detection in streaming graphs," in *Proceedings of the 29th ACM SIGKDD conference on knowledge discovery and data mining*, in KDD '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 93–104. doi: 10.1145/3580305.3599504[30].

[42] P. Liu *et al.*, "Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks," in *2020 IEEE 31st international symposium on software reliability engineering (ISSRE)*, 2020, pp. 48–58. doi: 10.1109/ISSRE5003.2020.00014[31].