POLITECNICO DI TORINO
Repository ISTITUZIONALE

MicroView: Cloud-Native Observability with Temporal Precision

Original
MicroView: Cloud-Native Observability with Temporal Precision / Cornacchia, A., Benson, T., Bilal, M., Canini, M.. - ELETTRONICO. - (2023), pp. 7-8. (CoNEXT '23: The 19th International Conference on emerging Networking EXperiments and Technologies Paris (France) 8 December 2023) [10.1145/3630202.3630233].

Availability:
This version is available at: 11583/2983046 since: 2023-12-21T10:13:54Z

Publisher:
Association for Computing Machinery (ACM)

Published
DOI:10.1145/3630202.3630233

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
ACM postprint/Author's Accepted Manuscript

(Article begins on next page)

08 October 2026

MicroView: Observability with Temporal Precision

Alessandro Cornacchia
Politecnico di Torino

Theophilus A. Benson
Carniege Mellon University

Muhammad Bilal
Unbabel

Marco Canini
KAUST

ABSTRACT

We present MicroView, a system designed to improve the accuracy and timeliness of observability in cloud-native applications, while minimizing overhead. MicroView stands out from conventional observability tools by incorporating local metrics processing stages at every node within a lightweight data-plane. We preliminary demonstrate the benfits for distributed tracing and outline a set of architectural choices focused on offloading the MicroView data-plane to IPU accelerators, such as BlueField2 SmartNIC, thus limiting the interference with running services.

CCS CONCEPTS

• **Networks** → Data center networks; **In-network processing**; • **Computer systems organization** → *Cloud computing*.

1 INTRODUCTION

Microservice observability is a key requirement for troubleshooting cloud-native applications, as it provides visiblity about their internal state. Observability tools collect a wealth of monitoring data — i.e., metrics, request traces and logs — which is then used to detect and diagnose failures and identify performance bottlenecks.

Unfortunately, observability can create a significant overhead on server resources thus creating resource contention and interference with user services. This overhead is mainly generated by data-copies and network stack processing [4] to communicate with the monitor backend. Even worse, it grows with the scale of monitored components and the frequency at which data is collected. In practice, operators need to resort to relaxed sampling rates for data collection, which sacrifices the quality of observability itself, such as accuracy and timeliness. Although scheduling ad-hoc CPU bonding for the observability processes or vertically scaling the infrastructure would mitigate the problem, these solutions are neither energy-efficient nor cost-efficient.

This work proposes a system to guarantee accuracy and timeliness of observability, while limiting the overhead and interference with running applications. Our design hinges on the observation that for metrics, the overhead is dominated by the *ingestion* costs rather than *generation* costs (Table 1). This allows us to — relatively cheaply — increase the temporal granularity at which new metrics samples are produced, and to decouple local generation rate from ingestion rate. In light of these observations, we address the benefits and challenges of *(1)* running a metrics processor on each node that can handle high metrics generation rate, and *(2)* exploiting such intelligence to locally extract actionable signals and assist performance debugging tasks. We take distributed tracing as a showcase example and demonstrate that MicroView can improve the coverage of anomalous requests (i.e., whose latency violates SLO) by approximately 5× compared to head-based sampling. Finally, we delineate architectural choices to offload the metrics processor to emerging Infrastructure Processing Units (IPUs) [5] accelerators, such as BlueField-3 SmartNIC.

2 PROPOSED DESIGN

In this section, we outline our proposed system architecture and discuss how it can be beneficial for the task of distributed tracing.

2.1 MicroView overview

The proposed system architecture is shown in Figure 1. Our system differs from traditional observability architectures [4] in that it adds metrics processor locally at each node. The ***data-plane*** is where metrics processing takes place. It consists of a bank of per-microservice classifiers. Each classifier periodically receives a metric vector relative to the corresponding microservice. It applies transformations (e.g., accumulation, data whitening, etc.), and looks for anomalies in the vector. As a classifier, we choose to adopt a streaming-based sketch algorithm [2], as it operates in single-pass without requiring storage of past samples, is lightweight to process high-rate streams and supports continual-learning from new samples at runtime. At its heart, the sketch learns a low-dimensional reconstruction basis for the data, and detects anomalies when it cannot reconstruct an input sample within a certain error tolerance. The ***control-plane*** consists of MicroView agents that orchestrates the interaction between the metrics sources (e.g., microservices) and the data-plane. If the data-plane sits on a IPU accelerator, MicroView agents ensures the IPU can access the memory regions where the metrics variables reside. Additionally, it produces actionable alerts to observability libraries based on sketch classifications. The specific action depends on the desired use-case, an example is provided in the next section.

2.2 Use-case: distributed tracing

Metrics reveal the internal state of applications and containers. Their unexpected variation potentially indicates an anomalous state,

Metrics observability configuration		**CPU usage at monitored node**
Generation rate [Hz]	*Ingestion rate [Hz]*	
1	1	12-13%
1	1/30	3-4%
1/30	1/30	2%

Table 1: CPU consumption of a single Kubernetes node, while monitoring metrics for the `Online-boutique` workload [1]. Ingestion rate is the frequency at which a Prometheus monitor — on a different node — queries metrics from the pods. Generation rate refers to the metrics creation/update frequency within the pods, and is controlled by changing `housekeeping_interval` in cAdvisor.

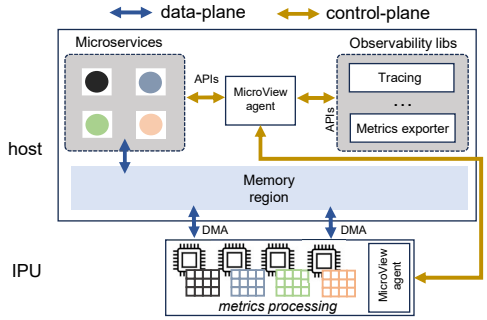

Figure 1: MicroView architecture. Unlike traditional observability architectures, we propose to add local metrics processing stages at each node.

which could negatively impact ongoing user requests. Timely analyzing metrics during runtime can anticipate anomalous executions and trigger distributed tracing. To achieve its goal, the MicroView agent combines outputs from different classifiers. If at least one classifier detects an anomalous state, the agent triggers the distributed tracer. At this point, the tracer samples all incoming requests until the next classification cycle. In this regard, MicroView substantially differs from traditional workflows, where traces and metrics are collected independently and correlated offline. Notably, MicroView is complementary to state-of-the-art Hindsights' retroactive sampling [6], as it can serve as a trigger to it.

3 PRELIMINARY EVALUATION

Experimental setup. We implemented an offline prototype in Python. It post-processes datasets of metrics and traces, collected from running a widely used benchmark application [1]. We ran experiments on a 4-node Kubernetes (v1.25.5) cluster, each equipped with 8 Intel Xeon E3-1230v6 CPUs at 3.50GHz, 32 GB of RAM and running Ubuntu 22.04. We deployed Istio service mesh on the cluster, and used Locust load generation. We instrumented for observability with Jaeger tracer and a Prometheus instance that collects service-level and container-level metrics every second. This interval is significantly lower than current production system practices [6]. Service-level metrics include Istio metrics and application metrics whenever available (e.g., Redis), while container-level metrics are cAdvisor resource usage counters (CPU, memory, disk I/O, etc.). We fit the sketch classifiers with an initial training phase. During this phase we also tune hyperparameter for each sketch (i.e., microservice) separately. First, we use a *healthy* metrics dataset from which initialize the sketches to learn "normal" behavior [2]. We obtain the dataset when the application runs in underload conditions and with over-provisioned vCPU and memory limits. Second, we inject faults in the service and produce a second dataset, that we use to select hyperparameters that gives best F1-score. We use ChaosMesh for fault injection, and we simulate stress scenarios on the containers, such as CPU and memory.

Can MicroView help tracing? We then use these pre-trained sketches to guide trace sampling (as per Sec.2.1). We randomly select a service every 30 seconds and inject on it a short failures of 5 seconds. We compare against head-based sampling in terms of coverage, i.e., percentage of collected anomalous traces, and overhead,

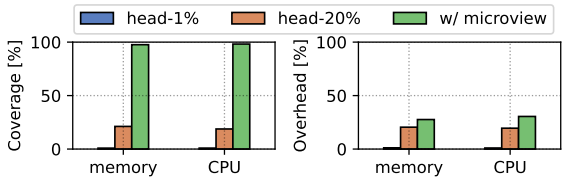

Figure 2: Performance-overhead trade-off of sketch-assisted distributed tracing for different anomalies.

measured as percentage of false positives. For us a trace is anomalous when exhibits tail latency or contains error codes. Figure 2 shows that tracing + MicroView achieves nearly total coverage, 5x better than sampling 20% of the request. This is because head sampling relies on luck to capture faulty traces, while MicroView is guided by metric signals. Its overhead is around 25% and generated by false alarms of the data-plane. We set the target of reducing it with metric selection [3] techniques.

4 RESEARCH AGENDA

We delineate several challenges we must address towards realizing our vision.

Host-IPU communication. The data-plane, that we plan to implement on a IPU accelerator, needs to access metrics that sit on the host OS [5]. A natural choice is to use DMA technology and bypass the host CPU. In this space, we identified two competing alternatives: RDMA and NVIDIA DOCA libraries. Their comparison is part of the future agenda. Complementary to it, the next step on the host-side is the definition and evaluation of the interfaces beween MicroView and microservices, for metrics creation and update.

Alternative use-cases. Different metrics have different nature. Some fluctuates on short time scales (e.g., CPU, power), other stays constant and change only in response to human reconfigurations. MicroView can work as a filter for metric ingestion, dynamically deciding which metrics are worth to ingest, which can save storage costs for the tenant and bandwidth for the provider.

REFERENCES

[1] 2023. Online-boutique. https://github.com/GoogleCloudPlatform/microservices-demo
[2] Hao Huang and Shiva Prasad Kasiviswanathan. 2015. Streaming anomaly detection using randomized matrix sketching. *VLDB Endowment* 9, 3 (2015), 192–203.
[3] Jörg Thalheim, Antonio Rodrigues, Istemi Ekin Akkus, Pramod Bhatotia, Ruichuan Chen, Bimal Viswanath, Lei Jiao, and Christof Fetzer. 2017. Sieve: Actionable insights from monitored metrics in distributed systems. In *ACM/IFIP/USENIX Middleware*.
[4] Zhe Wang, Teng Ma, Linghe Kong, Zhenzao Wen, Jingxuan Li, Zhuo Song, Yang Lu, Guihai Chen, and Wei Cao. 2022. Zero Overhead Monitoring for Cloud-native Infrastructure using RDMA. In *USENIX ATC*.
[5] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. 2023. Characterizing Off-path SmartNIC for Accelerating Distributed Systems. In *USENIX OSDI*.
[6] Lei Zhang, Zhiqiang Xie, Vaastav Anand, Ymir Vigfusson, and Jonathan Mace. 2023. The Benefit of Hindsight: Tracing Edge-Cases in Distributed Systems. In *USENIX NSDI*.