eBPF-Enhanced Complete Observability Solution for Cloud-native Microservices

Bhavye Sharma and Deepak Nadig
Dept. of Computer and Information Technology, Purdue University, West Lafayette, IN 47907
Email: {sharm609, nadig}@purdue.edu

***Abstract*—Microservices have emerged as a popular pattern for developing large-scale applications in cloud environments for their flexibility, scalability, and agility benefits. Furthermore, orchestration services like Kubernetes have simplified the deployment of cloud-native applications. However, monitoring and debugging these complex networked applications has become increasingly challenging, creating additional overheads. Traditionally, observability or monitoring requires developers to instrument their applications to expose metrics, logs, and traces using language-restricted libraries. This approach does not work well in a multi-tenant cloud environment as it cannot monitor processes or containers that are not instrumented or hidden. A critical challenge is managing complexity by consistently instrumenting multiple microservices across application platforms and programming languages. Hence, there is a need for a low-overhead cloud-native solution that provides complete observability for distributed and containerized environments. eBPF is a Linux VM technology that can instrument the host kernel directly and provides out-of-the-box cloud-native observability with negligible performance overheads. This paper proposes an eBPF-based solution that offers complete observability for cloud-native deployments. Further, we compare the performance and effectiveness of our solution with popular observability agents like Node Exporter and cAdvisor. We show that our proposed solution reduces CPU overheads by up to 210 times while requiring up to 159% less memory than the alternatives. Lastly, we deploy and test our solution on a Chameleon cloud bare metal testbed.**

***Index Terms*—eBPF, Observability, Kubernetes, Service Mesh**

I. Introduction

Observability in cloud-native applications is the ability to compile three vital classes of information, i.e., metrics, logs, and traces [1]. Metrics are numeric data representations over time and may utilize mathematical modeling or prediction to derive knowledge of a system's behavior [1]. Metrics are optimized for storage, compression, and retrieval, enabling longer retention and faster querying. Logs are an immutable, timestamped record of discrete events over time. Lastly, traces represent the interaction amongst distributed applications and enable a macro view of the request-response life-cycle. A single request may traverse multiple services spread across different servers or geographical locations in a distributed system. While achieving observability in monolithic applications is straightforward as their code-base/deployment resides on a single infrastructure endpoint, accomplishing the same task is complex in cloud-native deployments. In monolithic applications, the developer can instrument the application or the host to collect observability data. However, in cloud-native deployments with numerous containers, distributed hosts, and various microservices implementation ecosystems, it is not feasible to manually instrument distinct microservices or operating system (OS) kernels with additional observability code. Thus, a critical problem in ensuring observability for cloud-native applications is the associated overhead to maintaining and updating observability code across diverse microservices implementation ecosystems.

The sidecar (i.e., a helper container running alongside the application container in a Kubernetes pod) approach for observability reduces the overhead of maintaining multiple applications by redirecting all observability functionality from the application container to a separate (sidecar) container. Typically, sidecars deployed alongside each microservice handle networking functions and perform L3, L4, and L7 network observability tasks. However, the sidecar approach to observability is limited to the collection of network metrics only, i.e., latency, traffic/error rates and traces. Therefore, as sidecars add to the container overheads on the cloud-native cluster, they have the drawback of utilizing valuable cluster resources, including computing, storage and networking. Further, while sidecars are particularly useful in detecting problems in microservices, they fail to provide precise root-cause analysis of the problem. For example, we can identify misbehaving services using traces and analyze latency, but debugging the application containers' internals is impossible via sidecars. Therefore, to achieve complete observability of containers, we must collect observability data directly from the low-level host OS. While numerous agent-based approaches exist (e.g., OpenTelemetry, cAdvisor [2], AWS X-Ray, Cloudwatch Container Insights), deploying these solutions (typically in user space) incurs additional resource overheads, impacting deployed workloads' performance.

This paper presents a complete observability solution that utilizes the extended Berkeley Packet Filter (eBPF) in the Linux kernel. Our proposed solution relies on small eBPF-based event-triggered programs to extend the OS functionality without instrumenting the corresponding application code. Unlike existing solutions, our eBPF-based approach operates in the kernel space at line speeds without degrading the performance of the events they are instrumenting. By leveraging kernel space eBPF-based observability agents, our solution directly collects deep contextual observability data without additional overheads. We demonstrate the effectiveness and superiority of our proposed solution through significant overhead

reduction compared to the status quo. Further, our proposed solution allows for deep contextual visibility into the observed application without the burden of instrumentation.

Our specific contributions are as follows:

- We evaluate current non-eBPF observability solutions and identify performance overheads and bottlenecks.
- We propose and implement an effective solution providing complete observability for using eBPF exporter [3] and Deepflow.io [4].
- We evaluate the scalability and effectiveness of our solutions using a real-world cloud test-bed deployment and assess it using chaos experiments and user workloads.

The rest of this paper is organized as follows: Section II provides the background and related works in observability and provides the context for our work; Section III describes the design of the microservice observability framework for cloud-native infrastructure; Section IV presents the solution architecture of our eBPF-enhanced microservice observability framework; Section V describes the experimental testbed and setup; Section VI presents extensive results from our experiments and the discussion; In Section VII we conclude our work and outline the future research directions.

II. Background and Related Works

This section presents an overview of Kubernetes, OpenTelemetry, and eBPF and outlines the related works and associated observability challenges.

A. Kubernetes and OpenTelemetry

Kubernetes [5] is a highly modular and extensible open-source container orchestration platform that automates containerized applications' deployment, scaling, and management. Maintained by the Cloud Native Computing Foundation (CNCF), its architecture consists of interconnected components that work together to manage container lifecycle and their resource consumption. OpenTelemtry (OTel) [6] is a CNCF-incubated project originating from Google's OpenCensus and OpenTracing projects. OTel provides standardized Software Development Kits (SDKs), Application Programming Interfaces (APIs), and tools for ingesting, transforming, and shipping data to an observability backend. OTel provides per-language instrumentation libraries to export metrics, logs, and traces from applications. The framework supports various languages and platforms, including Java, Go, Node.js, Python, and .NET. Instrumenting each application with OTel is tedious in an agile, multi-language environment. Another limitation of OpenTelemetry is that it is only aware of the L7 metrics of an application; it does not collect information in other layers, the networking stack or the kernel level, which is critical for cloud-native observability. Lastly, OpenTelemetry clusters observability data from multiple pods together as services, limiting its ability to perform pod-level analysis.

B. Extended Berkeley Packet Filter (eBPF)

Recently, eBPF has garnered much attention for load balancing, firewalls, and network security [7]. For example, Android uses it for in-kernel tracing and security [8], and Netflix uses it for injecting observability data into their networking stack. The original BPF patch [9] added a new programmable virtual machine (VM) with two registers to filter networking events based on predicates. In 2013, Alexei Starovoitov patched this VM with eight 64-bit registers and added support for kernel probes (*kprobes*), user probes(*uprobes*), and user statically defined traces (USDT), which extended the capabilities of BPF beyond network filtering. An eBPF program is a sequence of 64-bit instructions verified for safety and high performance by the compiler. It is JIT-compiled using tools like Clang LLVM on the host machine to instrument it for the target. eBPF programs export kernel-level metrics by writing them to memory maps located inside the kernel space but accessible via user space. This approach reduces the need to copy data from kernel space to user space, allowing for higher throughput at lower overheads. This lower overhead allows us to dynamically run eBPF scripts in production environments to debug real-world systems without impacting users.

C. Related Works

The authors in [10] propose ViperProbe, an eBPF-based deep microservice metrics collection tool. ViperProbe focuses on improving metrics collection for service meshes by associating Process ID (PID) context with the metrics. This approach improves over generic metrics monitoring that does not differentiate individual host services. ViperProbe also includes an offline search paradigm for analyzing patterns to determine the minimal but effective set of metrics, i.e., CriticalMetrics, that enable run-time diagnosis of a service. In [11], the authors create a microservice tracing system requiring minimal code modification (auto instrumentation) and demonstrate tracing with little influence on the system performance. Their system monitors network packet headers and generates traces but does not consider other in-kernel metrics. The authors in [12] propose MiSeRTrace, a kernel-level microservice visibility tool that utilizes eBPF to instrument tracing for individual pods in a service mesh at the kernel level. The work in [13] uses the eBPF to perform profiling using eBPF in a distributed payments system using two implementations of the Interledger protocol specifications. In contrast to the above, our work focuses on developing an eBPF-based, complete observability solution for cloud-native environments with metrics,logs and distributed traces.

Unlike monolithic architectures, where components communicate through in-process calls, microservice architectures encounter more service interaction failures as they happen over potentially unreliable networks. Containers are designed to be ephemeral, spun up and destroyed rapidly, making it challenging to track their associated data. Also, containers deployed on distributed hosts make gathering a comprehensive view of service behavior and performance complex. Lastly, using Kubernetes to manage containers necessitates monitoring additional components such as the `kube-apiserver`, `kube-controller-manager`, proxies, `kubelet`, and

the container network interface (CNI). Next, we present our eBPF-based microservices observability framework.

III. Proposed Observability Framework

Current solutions to achieve observability utilize user space programs to capture and analyze data, resulting in significant performance overheads. Our proposed approach focuses on moving these responsibilities to kernel space eBPF programs to drastically reduce the overheads while attaching deep contextual information for this data from the kernel. Further, we can dynamically inject our proposed eBPF framework into any deployed application without recompilation or redeployment. Our proposed solution improves over existing eBPF exporters from Cloudflare and incorporates new features into several `.bpf.c` scripts, including `accept-latency`, `cachestat`, `llcstat`, `malloc`, `oomkill`, `runqlat`, `shrinklat`, and `tcpbacklog`.

Our proposed microservice observability framework employs an eBPF-based solution to monitor the following three layers of containerized applications in cloud-native settings:

1) *Microservices Layer*: Using OTel libraries, we instrument microservices to attach additional distributed trace context to the *East-West* network packets. The context is propagated in the headers of network packets.
2) *Container Network Layer*: Using our proposed eBPF agents deployed on our cluster nodes, we monitor our applications' RED metrics, i.e., rate error and duration. The eBPF agents captured and exported throughput, latency and error codes. By leveraging the Deepflow eBPF library [4], the agent provides a mechanism for the automated collection of context-propagation and OpenTelemetry distributed traces.
3) *Infrastructure Layer*: Our proposed eBPF agents deployed with Cloudflares' eBPF exporter enable us to obtain detailed container-aware telemetry about the cluster nodes and aid in debugging and troubleshooting anomalies.

Our framework relies on Prometheus to provide a data store for observability data sources and helps visualize it using Grafana [14] dashboards. Figure 1 shows our proposed frameworks' integration with existing Kubernetes infrastructure. Our proposed solution consists of multiple eBPF programs running in the `kernel-space` triggered by system calls generated by our application. A prometheus collector runs on worker nodes and regularly scrapes the eBPF agents on our cluster to extract data the eBPF agent collects.

IV. Solution Architecture

Our solution achieves complete microservices observability for Kubernetes applications by combining three observability levels. Further, our solution ensures scalability and efficient operation to collect metrics and traces from distributed containers and detect performance anomalies. We deploy our eBPF-based solution on each cluster node and extend the OS functionality dynamically without affecting the application life-cycle. We export observability data generated at all

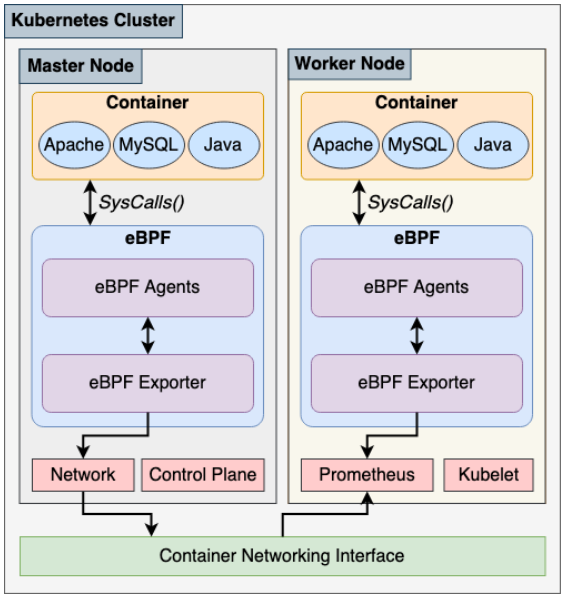


Fig. 1: eBPF-based observability agents

levels to a Prometheus instance and visualize them using our custom-built Grafana dashboards. Figure 2 shows the internal architecture of our worker nodes. Our microservices run inside the container runtime environment behind an Envoy sidecar proxy. The eBPF agent runs as a DaemonSet in all our cluster nodes `kernel-space`. The sidecar, collector and applications provide distributed trace IDs and metrics to the Prometheus instance with a scrape interval of 1 second.

Algorithm 1 Measuring I/O Operations Latency

Require: Number of Disks attached to kernel (D_n), Time of Operations Hash Map ($H_{t_{start}}$).
Output: F(`block_rq_insert`)$\leftarrow$ `kprobes`, and F(`block_rq_issue`) $\leftarrow$ `kprobes`.

$H_{t_{start}} \leftarrow$ Record the current timestamp for request r_i.
if `block_rq_complete` $=$ True **then**
  $\Delta t(r_i) = t_{now} - t_{start}$
end if
Prometheus $\leftarrow$ eBPF Exporter $\leftarrow H_{t_{start}}$

Algorithm 2 Mapping Distributed Network Traces

Require: Trace Information Hash Map (H_{DT}).
Output: F(`connect`) $\leftarrow$ `kprobes`,
F(`accept`) $\leftarrow$ `kprobes`, and
F(`close`) $\leftarrow$ `kprobes`.

if $Num_{Bytes} \neq 0$ **then**
  $trace_{id} \leftarrow$ `x-Request-ID`
  $H_{DT} \leftarrow trace_{id}\{\langle src, dest, timestamp, proto\rangle\}$
end if
if Exists $trace_{id}$ **then**
  /* Create full span */
  Append $\langle src, dest, timestamp, proto\rangle$ to $trace_{id}\{\}$
end if
Prometheus $\leftarrow$ eBPF Exporter $\leftarrow H_{t_{start}}$

The Algorithms 1 and 2 shown below present two example observability operations implemented by our solution. Algorithm 1 outlines the procedure to compute the block I/O latencies using the eBPF hash map, and Algorithm 2 demonstrates the procedure for creating complete time spans by mapping distributed network traces using a $trace_{id}$.

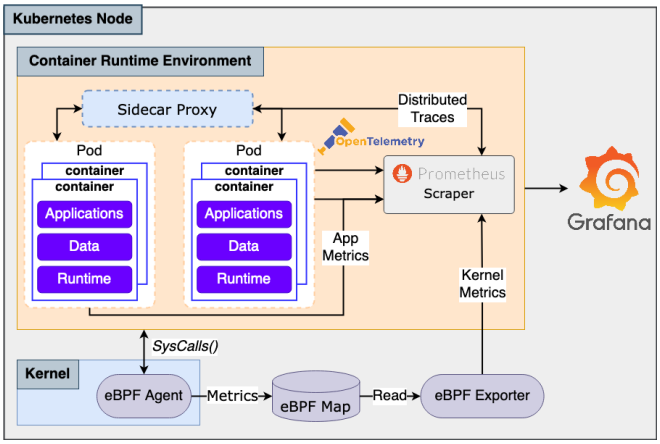


Fig. 2: Proposed Solution Architecture

A. Reducing OpenTelemetry Collector Overheads

Manually instrumented applications are the first level of observability in our solution. We achieve this by integrating OTel libraries to instrument our application. We propose a two-step approach to achieve complete application-layer observability. First, we use the OTel language SDK to generate metrics, logs, and traces. While this requires the application developer to instrument an application, it is a robust way to expose custom (organization-specific) observability data. Next, our eBPF-agent collects the metrics and traces generated by the OpenTelemetry instrumented application and exports them to Prometheus. Our eBPF-based solution leverages the applications' port ID or process ID as a key for the eBPF map to gather data.

B. Eliminating Sidecars for Distributed Tracing

Kubernetes microservice architecture utilizes an ingress or reverse proxy (Envoy [15]) as the entry point for all incoming network packets in a network service mesh. OpenTelemetry uses this proxy to generate distributed traces for the applications. When pods forward requests to other pods in our network, the pods' Envoy sidecars extract and propagate the distributed trace ID using the `X-Request-ID` header. By aggregating information from multiple sidecars, we visualize the round-trip of our request-response cycle in a distributed trace. Envoy's sidecar approach also tracks metrics such as throughput, latency, and errors. Sidecars like Envoy help provide layer-7 observability due to their ability to inspect network packets and decrypt packets using transport layer security (TLS). However, a complete network stack observability solution does not require deploying a user space sidecar. Our proposed solution leverages Deepflow [4] to demonstrate an eBPF-based approach to distributed tracing in kernel space. Our solution uses eBPF to natively parse all packets flowing in our network and extract the `X-Request-ID` header to generate a complete distributed trace.

C. Improving the Accuracy of Host-level Metrics

Most host-level observability agents are privileged applications that access the `/proc` virtual file system. The `/proc` folder contains runtime system information [16] (e.g., hardware configuration). Prometheus NodeExporter is a wrapper that reads data from the `/proc` folder and serves it on an HTTP endpoint. Unlike NodeExporter, cAdvisor [2] exports metrics and provides a per-container resource utilization breakdown, i.e., a container-aware exporter. While NodeExporter and cAdvisor provide time series metrics by sampling data in the `/proc` folder, they result in information loss [17] based on the choice of the sampling interval. Our proposed eBPF-based solution takes a novel approach to the sampling problem. Our eBPF agent works by running eBPF programs within the Linux kernel and collects metrics directly from the kernel. These metrics include comprehensive system performance, resource utilization, and network traffic information. These metrics are written to an eBPF Map, capturing all events without information loss. Further, using an eBPF exporter results in significantly low overhead ($< 1\%$ of NodeExporter [3]), and ensures comprehensive metrics collection. Thus, our solution allows access to metrics at the kernel level that are inaccessible by other tools. This approach makes over 2000 metrics and tracepoints available for the kernel. Table I compares the observability capabilities of OpenTelemetry, Envoy Sidecar, cAdvisor, and NodeExporter with our proposed solution.

D. Test Application Deployment

To test the efficacy of our solution, we deploy a complete microservice-based e-commerce portal [18] with ten unique services. Using locust.io, a load-testing framework, we generate API calls to services and simulate real-world workloads. Our test setup integrates applications developed on multiple programming languages, including TypeScript, Go, Javascript, C++, and C#, and is instrumented to export OpenTelemetry metrics, logs, and traces. Figure 3 shows the service graph for the 14 microservices in our deployment. The edge weights represent the number of requests between the microservices. We utilize the load generator (Locust) to simulate up to 1000 concurrent users at a spawn rate of 20 users per/second to generate service traffic for our frontend proxy (Envoy).

V. Experimental Setup

We use the Chameleon Cloud bare-metal platform to evaluate our eBPF-based observability solution on a 3-node (1 master, 2 workers) Kubernetes cluster. Each bare-metal node instance provides an Intel(R) Xeon(R) E5-2670 v3, 2-socket CPUs with 48-threads, 128GB of RAM and InfiniBand networking. We provision the nodes with Ubuntu 22.04, and our solution will work on any OS with Linux kernel v5.15 or newer. We employ Containerd v1.6.12 as the container runtime and deploy Kubernetes v1.25.6. A 100GB NFS share

	OpenTelemetry	Envoy	cAdvisor	NodeExporter	Our Solution
Traffic Load	×	✓	✓	✓	✓
Packet Loss	×	✓	✓	✓	✓
HTTP Errors	✓	✓	✓	×	✓
Socket Metrics	×	✓	✓	×	✓
Latency	×	✓	×	×	✓
Container Health Check	×	✓	✓	×	×
Container Lifecycle Events	×	×	✓	×	✓
Memory Utilization	×	×	✓	✓	✓
CPU Utilization	×	×	✓	✓	✓
Disk I/O	×	×	✓	✓	✓
File System Utilization	×	×	×	✓	✓

TABLE I: Comparison of our Proposed Solution with other Observability Tools

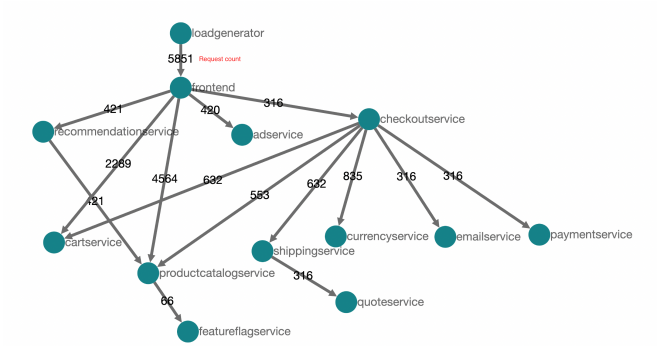


Fig. 3: Test Application Service Graph with Microservices

stores the Prometheus metrics generated by our cluster. We deploy various observability tools on our cluster, including OpenTelemetry Collector, Envoy proxy, NodeExporter, and cAdvisor and compare them to our eBPF-based solution. Our eBPF agent leverages the eBPF exporter for in-kernel metrics collection and Deepflow for distributed microservices tracing.

VI. Results and Discussion

First, we evaluate the performance of our solution and compare our approach with existing solutions for application-level observability. Next, we assess the efficiency of our solution and provide insights on reducing the latency associated with eliminating sidecars for distributed tracing. Lastly, we present workload tests to show our solutions' performance in a real-world setting and compare them with the alternatives.

A. Evaluating Observability Overheads

We compare our proposed eBPF solution with popular observability tools, including OpenTelemetry, Envoy, cAdvisor, and Node Exporter, as shown in Figure 4a. We observe that our solution consumes marginally more memory ($\sim 12\%$) than the most memory-optimized tool (i.e., NodeExporter). However, it consumes between $30.95\% - 159.5\%$ less memory than the alternatives. Further, we also observe that our eBPF-based solution consumes between $21\times - 214\times$ fewer CPU resources than the alternatives.

To evaluate I/O performance, we employ Flexible IO Tester (Fio) to spawn several threads to help us test a particular type of I/O action the user performs. We conduct random read/write and sequential read/write operations on our worker nodes and used our solution to detect these workloads. Based on our observations, host-level metrics were the best indicator for I/O performance issues. Unlike other tools, our solution implements a block I/O latency measurement (See Algorithm 1) mechanism that collects detailed metrics (and identifies longer read and write latency) and histograms. As eBPF scripts can store some information in the eBPF map, they can aggregate latency information locally and present it in a histogram format when Prometheus scrapes the `/metrics` endpoint.

B. Latency Performance

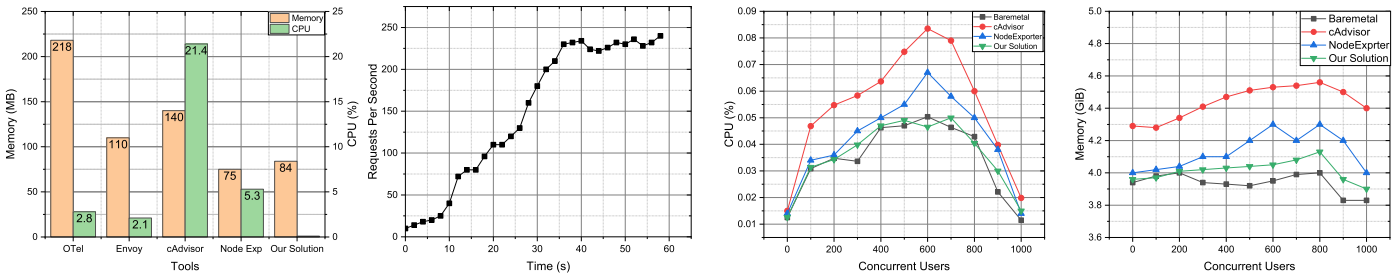
Traditional distributed tracing, configures Envoy to send tracing data to the OpenTelemetry Collector, and uses the Collector's HTTP receiver to send data. The Collector is configured to listen on a specific endpoint, i.e., `/v1/trace` for trace data and waits for incoming requests. When Envoy generates a trace, it sends an HTTP POST request to the Collector's HTTP receiver, with the trace data in the request's body. From our testing, we observed that, on average, there was a $250\mu s$ blocking POST request from Envoy to OpenTelemetry whenever it attempts to create a distributed trace.

Let $P_{(t,i)}$ denote the latency for each (i^{th}) HTTP POST request to the `/v1/trace` endpoint. In a complete service mesh architecture with multiple Envoy proxies, we incur a latency cost of $\sim 250\mu s$ for *every proxy traversed* by the service request in that request path. Thus, the total latency for a given service request (with $N = \{1, ..., n\}$ proxy traversals) is given by:

$$P_t = \sum_{k=1}^{n} \alpha_k P_{(t,k)} \tag{1}$$

where α_k denotes the latency variation factor for the k^{th} hop.

Our solution eliminates the need for this POST request by programming the kernel to extract trace-ids from packets and correlating data using the Deepflow library. This has significantly faster performance since it is avoiding a blocking



(a) Overhead comparison of observability tools.

(b) Requests per second generated during the Locust test.

(c) Workload testing cluster CPU utilization comparison.

(d) Workload testing cluster memory utilization comparison.

Fig. 4: Experimental Results.

call to the OpenTelemetry collector for every packet traversing the request path.

C. Workload Performance Testing

We use a locust.io script to generate HTTP calls to our application for workload performance tests. Current distributed tracing solutions forward all incoming requests to the Envoy proxy; Envoy adds trace ID information to these requests, allowing us to track them in our network service mesh. We observed increased response times during our workload tests with the Python HTTP library locust.io employs. While we can improve response times by leveraging HTTP2 and increase throughput, we cannot eliminate the HTTP blocking call latency itself. Our Locust script generates requests from 1000 users, as shown in Figure 4b. Figures 4c and 4d present the cluster CPU and memory utilization during our workload testing. The figure shows that our eBPF solution adds minimal overhead to the bare metal benchmark, i.e., with no observability tool deployed (baseline). cAdvisor by Google, on the other hand, had the highest overheads as indicated by the CPU and memory utilization data. From Figures 4c and 4d, we also observe that the CPU utilization peaks at 600 concurrent users since Locust requires additional resources as it ramps up the requests per second but stabilizes eventually, as shown in Figure 4b.

VII. Conclusions

This paper presents an eBPF-enhanced complete observability solution for microservices deployed on distributed Kubernetes clusters. Our work focuses on developing a holistic observability engine for Kubernetes workload that combines the microservices, container network, and infrastructure telemetry data into a unified framework. Further, we highlight the complex interactions between modern applications and their monitoring challenges. In addition, we developed a metrics and trace collection framework that eliminates the burden of developing and maintaining instrumented applications. We conducted extensive experiments, and our evaluations show that the proposed solution effectively reduces resource overheads compared to the alternatives. Specifically, our solution reduces cluster CPU and memory resource utilization, on average, between $21-214$ times and $30\%-159\%$, respectively. Our eBPF-enhanced solution allows us to monitor metrics at the host level and provide highly detailed information not attainable by other tools. Lastly, our approach eliminates the need for HTTP blocking calls for distributed tracing and significantly improves latency performance.

References

[1] C. Sridharan, *Distributed Systems Observability: A Guide to Building Robust Systems*. O'Reilly Media, 2018.

[2] Google, "cAdvisor," https://github.com/google/cadvisor, November 2014, accessed on May 1, 2023.

[3] Ivan Babrou, "Introducing eBPF Exporter," https://blog.cloudflare.com/introducing-ebpf_exporter/, February 2022, accessed on May 1, 2023.

[4] J. Shen, H. Zhang, Y. Xiang, X. Shi, X. Li, Y. Shen, Z. Zhang, Y. Wu, X. Yin, J. Wang, M. Xu, Y. Li, J. Yin, J. Song, Z. Li, and R. Nie, "Network-centric distributed tracing with deepflow: Troubleshooting your microservices in zero code," in *Proceedings of the ACM SIGCOMM 2023 Conference*, ser. ACM SIGCOMM '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 420–437. [Online]. Available: https://doi.org/10.1145/3603269.3604823

[5] CNCF, "Kubernetes," https://kubernetes.io/, accessed on May 1, 2023.

[6] CNCF, "OpenTelemetry," https://opentelemetry.io/, Accessed 2023.

[7] Facebook, "Katran," https://github.com/facebookincubator/katran, May 2018, accessed on May 1, 2023.

[8] Google, "Extending the Kernel with eBPF : Android Open Source Project," https://source.android.com/docs/core/architecture/kernel/bpf, accessed on May 1, 2023.

[9] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," in *USENIX winter*, vol. 46, 1993.

[10] J. Levin and T. A. Benson, "ViperProbe: Rethinking Microservice Observability with eBPF," in *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*, 2020, pp. 1–8.

[11] M. Song, Q. Liu, and H. E., "A Mirco-Service Tracing System Based on Istio and Kubernetes," in *2019 IEEE 10th International Conference on Software Engineering and Service Science*, 2019, pp. 613–616.

[12] Thrivikraman, V. R. Dixit, N. R. S, V. K. Gowda, S. K. Vasudevan, and S. Kalambur, "MiSeRTrace: Kernel-level Request Tracing for Microservice Visibility," 2022. [Online]. Available: https://arxiv.org/abs/2203.14076

[13] C. Cassagnes, L. Trestioreanu, C. Joly, and R. State, "The rise of eBPF for non-intrusive performance monitoring," in *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2020, pp. 1–7.

[14] Grafana Labs, "Grafana," https://grafana.com/, accessed on May 1, 2023.

[15] Lyft, "Envoy," https://www.envoyproxy.io, accessed on May 1, 2023.

[16] The Linux Documentation Project, "The /proc Filesystem," https://tldp.org/LDP/Linux-Filesystem-Hierarchy/html/proc.html, 2004, accessed on May 1, 2023.

[17] L. Jackson, "Debugging microservices: Lessons from google, facebook, lyft," https://thenewstack.io/debugging-microservices-lessons-from-google-facebook-lyft/, 2017, accessed on May 1, 2023.

[18] Opentelemetry, "OTEL Demo," https://opentelemetry.io/ecosystem/demo/, accessed on May 1, 2023.