ViperProbe: Rethinking Microservice Observability with eBPF

1st Joshua Levin
Brown University

2nd Theophilus A. Benson
Brown University

***Abstract*—Recent shifts to microservice-based architectures and the supporting *servicemesh* radically disrupt the landscape of performance-oriented management tasks. While the adoption of frameworks like Istio and Kubernetes ease the management and organization of such systems, they do not themselves provide strong observability. Microservice observability requires diverse, highly specialized, and often adaptive, metrics and algorithms to monitor both the health of individual services and the larger application. However, modern metrics collection frameworks are relatively static and rigid.**

We introduce ViperProbe, an eBPF-based microservices collection framework that provides (1) dynamic sampling and (2) collection of deep, diverse, and precise system metrics. ViperProbe builds on the observation that the adoption of a common set of design patterns, e.g., servicemesh, enables offline analysis. By examining the performance profile of these patterns before deploying on production, ViperProbe can effectively reduce the set of collected metrics, thereby improving the efficiency and effectiveness of those metrics.

To the best of our knowledge, ViperProbe is the first scalable eBPF-based dynamic and adaptive microservices metrics collection framework. Our results show ViperProbe has limited overhead, while significantly more effective for traditional management tasks, e.g., horizontal autoscaling.

I. Introduction

Microservices are the result of a series of evolutions in distributed systems aimed to design more abstract, lightweight, flexible, and scalable systems for cloud platforms. The growth and rapid adoption of tools like Docker [1] and Kubernetes [2] quickly made container-based design an industry standard. These tools made deploying, managing, and developing microservice architectures tractable for companies. Following the growth of microservices, a series of design patterns and frameworks for managing large microservice deployments emerged. These patterns and tooling (i.e. Istio [3] and Linkerd [4]) are referred to as the *servicemesh*. The resulting servicemesh has significantly increased the velocity of code changes and deployment, the diversity and specialization of services, and the required coordination between services. The extreme number, heterogeneity, diversity and code-velocity of microservice components (i.e., services) significantly challenges traditional diagnosis and troubleshooting techniques [5], [6], [7], [8], [9].

In distributed systems, observability describes the ability to understand *what*, *where*, *when*, and *why* events took place in order to perform performance management, optimization, or debugging. Observability in distributed systems relies on three key tools: distributed tracing, metrics, and logs. Microservices dramatically increase the diversity in and magnitude of the metrics within the systems [10], [11]. As such, there has been significant work [12], [13] aimed at understanding which subset of metrics are relevant for each task. It is generally understood that only a subset of metrics and traces are relevant for each performance management task (e.g. scaling, overload control, routing). These metrics which are central for performing management tasks effectively are the *CriticalMetrics*. The growing wisdom is that the constant code changes and number of metrics and components make offline analysis intractable, and the deluge of data requires online analysis and sampling of metrics.

In this paper, we present ViperProbe, an alternative approach and platform for determining and instrumenting the CriticalMetrics for microservices. We build on the observation that while microservices are extremely diverse, the underlying infrastructure frameworks, e.g., servicemeshes, introduces significant uniformity across the system. Specifically, many microservice deployments have adopted microservice design patterns (Section II-B), which are in fact, more static and stable than the constantly evolving codebases of microservices. The static, shared nature of these components makes them more amenable to offline analysis thereby reducing the complexity and overhead of online techniques.

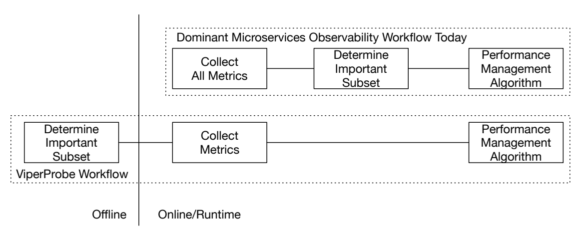

Figure 1. Paradigms for Observability Today

In this work, we take the first step towards this vision by presenting a framework, ViperProbe, an adaptive eBPF metrics collection tool for microservices. ViperProbe's eBPF metrics provide deep visibility into system performance characteristics, and ViperProbe collects and monitors metrics at the service level. In this paper we only demonstrate a subset of applications and leave the exploration of others for future work. There are two challenges in realizing ViperProbe: first, determining the CriticalMetrics is contingent on the specific combinations of microservices patterns, performance algorithms, and workloads. Thus, given the same container pattern but with different workloads or performance methodologies, we can expect different CriticalMetrics because the different workloads will exercise the code in different ways, and performance algorithms will interpret metrics differently.

Second, today, metrics collection is relatively rigid (Figure 1 Top); microservice frameworks usually collect all possible metrics, exposing them into an observability framework, e.g., Grafana, and then the performance algorithms determine the subset to use for analysis [13], [12], [14], [15], [16]. However, with our approach (Figure 1 Bottom), we plan to collect only a subset of the metrics initially with dynamic adjustments in real-time.

To tackle these challenges, ViperProbe builds on the flexibility of eBPF to dynamically enable and sample metrics. To the best of our knowledge, ViperProbe is the first scalable eBPF-based dynamic and adaptive microservice metrics collection framework. ViperProbe includes an offline search paradigm for analyzing patterns to determine the minimal but effective set of metrics, i.e., CriticalMetrics, for enabling runtime diagnosis of a service. ViperProbe uses the results of the offline analysis, i.e., the CriticalMetrics, to determine which metrics to activate initially. Additionally, ViperProbe was also designed to support online techniques, though, we leave the exploration of combined online and offline techniques for future work.

Our evaluation shows ViperProbe has between 10-15% CPU overhead running our entire set of implemented metrics. For latency overhead, our results show between 40-60% latency overhead at the 50th percentile, with negligible tail overhead at higher percentiles. Lastly, in our experimental application of ViperProbe for horizontal autoscaling, we found that ViperProbe greatly reduced failure rates (median reduction of 67%, Figure-7) using a subset of CriticalMetrics determined via k-Shape clustering.

II. Background

In this section we present an overview of microservice and servicemesh architectures, discuss their design components, and outline observability challenges for them.

A. Microservices

A microservice architecture is a loosely-coupled highly distributed system with individual, small services communicating through shared libraries or tooling. The microservice design philosophy is centered around independent, lightweight, and highly modular services. Several companies (Amazon [17], Microsoft [18], Facebook [19], Twitter [7], Lyft [19], [20], Uber [21], Netflix [8], Airbnb [6]) have adopted microservice patterns primarily for the following benefits:

1) Failure, resource, dependency, environment isolation
2) Greater abstraction with stricter APIs between services
3) Independent scaling, development, deployment, and replication of services

The result is a set of highly heterogeneous services running a polyglot of languages, with high velocity development and deployment [22], [23], [24], [5].

B. Microservice Design Patterns

To achieve the loose-coupling and coordination, developers developed *microservice design patterns* to simplify microservice development. Unlike traditional code design patterns, which are guidelines and rules for writing code, microservices patterns are in and of themselves code components – in certain cases, the patterns are services themselves, e.g., DB-patterns – Redis or Postgres [25], [26]. Next, we highlight the top three patterns with the goal of demonstrating the role of these patterns rather than providing a complete taxonomy.

In Figure 2, we present a canonical microservice deployment to illustrate the role of design patterns in modern microservices. The gray boxes are developer code, and the orange boxes are microservice design patterns.

Gateway Gateways are used to provide uniform access and control to these internal services without requiring separate teams to implement ingress/egress themselves. Clients issue requests to replicated Gateways who then pass those requests directly to services. Gateways provide authentication (Open Policy Agent), encryption (TLS, mTLS), traffic management, and observability for microservices [25]. Two popular Gateways are Envoy [27] (configured to be front-edge) and Ambassador [28].

Sidecar/Proxy The sidecar proxy pattern is an isolated, colocated process that runs alongside each microservice [29]. The proxy redirects all external network communication through it in order to provide serialization, security, or encryption [30]. By using proxies, independent teams can develop their services using the POSIX network stack and then deploy alongside the proxy. The proxy then can communicate with other proxies, thus enabling inter-service communication. A popular sidecar proxy, developed by Lyft and used by Istio, is Envoy [27] and is the primary proxy explored in this paper.

Servicemesh The *servicemesh* pattern [3], [4] is a specialized instance of the sidecar pattern which provides communication, discovery, security, traffic management, observability, and replication. The servicemesh abstracts the microservice design from the underlying network infrastructures and provides a set of common functionality required to stitch together distributed components, e.g., authentication and discovery. In essence, servicemesh frees microservice developers from having to rewrite this common functionality.

Takeaway: Unlike developer code (i.e., gray boxes), the patterns (i.e., orange boxes) are more static and more rigid. To illustrate this point, in Figure 3, we present CDFs of the time between releases for several patterns. We observe that while the different patterns have different release frequencies, they are often released every few weeks, which is radically different from studies that show that developers push changes to their microservice codebase multiple times a day [22], [23], [24]. This static and rigid codebase is more amendable to offline analysis because of its more gradual updates. We also note that some patterns, the Operator, in particular, already have bodies of work. Thus, translating offline performance of these patterns to the servicemesh is easier than that of their

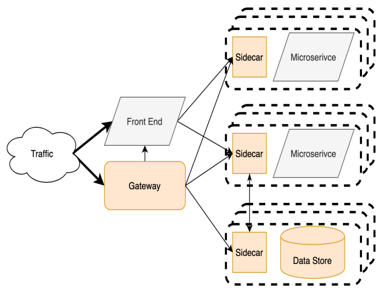

Figure 2. Canonical Microservices deployment

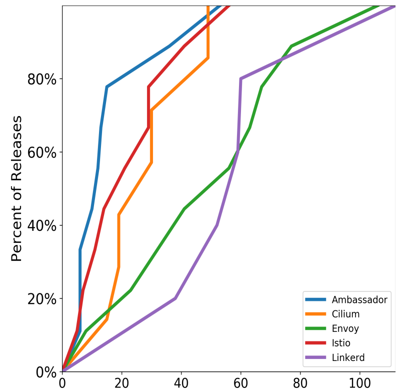

Figure 3. Days between Releases

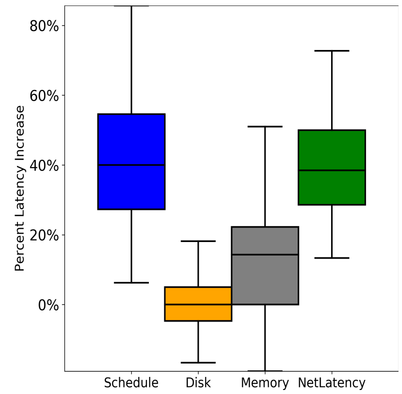

Figure 4. Example Application

underlying services. Lastly, we recognize the density of these patterns, in particular the sidecar pattern, enables such offline analysis to apply to large portion of deployed code.

C. Observability Challenges

The core fundamental components of observability in distributed systems are tracing [31], [32], [33], [34], [35], metrics [13], [15], [14], [16], and logging [36]. The challenges for distributed observability are collecting data at-scale and applying semantics to the data that enables actionable inferences. At two major servicemesh adopters, Netflix monitors more than 1.2 Billion [10] unique metrics and Uber monitors more than 700 Million [11]. The servicemesh introduces further challenges for microservice observability, some of which we highlight here:

1) Increased diversity of the services increases the variety of instrumentation and resulting metrics
2) The extreme hyperscaling of microservices explodes the volume of metrics, traces, and logs
3) Complex request paths and routing makes localizing, and qualifying ”normal” performance characteristics exponentially more challenging

Today, tracing is largely used for localization, which allows DevOps (Developer-Operators) to narrow their focus to a subset of metrics and logs to analyze. In this work, our main focus lies in metrics — more specifically, making metrics collection more dynamic and adaptive. The more common way to tackle the microservice challenges is to sample the data. Unfortunately, sampling leads to a loss of information and, thus, DevOps may be unable to detect problems [19]. Given this loss of information, when DevOps detect a problem, they turn off sampling. In fact, many production-grade monitoring systems provide a special “watershed” mode where the DevOps can turn off sampling and collect unsampled data [19].

D. Not All Metrics are Equal

Intuitively, the notion that a subset of metrics are more important than others is not a fundamentally new idea. However, most contemporary approaches [12], [15], [13], generally capture all metrics and then determine the important subset to analyze at runtime. A key often overlooked fact is that the overheads of metrics collection is a function of the type, number, and instrumentation for the collected metrics. In the area of microservices, this is especially relevant. The extreme diversity of services results in a polyglot of metric tooling thereby increasing metric complexity. Thus, being able to narrow and limit the metrics collected to a subset can be beneficial for performance. To illustrate this point, in Figure 4, we classify eBPF metrics collected by ViperProbe and present their overheads. The key observation is that while some metrics can be “always-on” due to their lightweight nature, e.g., Disk-related metrics, other classes of metrics are prohibitively expensive to constantly collect, e.g., network or scheduling. In Section-III-A, we further discuss how not all metrics are equally *relevant* to particular services beyond their inherent overhead differences.

E. extended Berkley Packet Filters (eBPF)

Linux Berkley Packet Filters have undergone extensive improvements in the recent years (Linux Kernel 3.15+ and 4.15+) bringing them to the forefront of kernel tracing and metric collection. Linux’s extended Berkeley Packet Filter (eBPF) feature enables developers to run small, static programs attached to kernel functions (kprobes), kernel tracepoints, or userspace functions (uprobes). Importantly, eBPF supports shared data structures between user and kernel space in order to pass information between programs and user processes. Facebook uses eBPF for TCP-Tuning, L-4 load balancing, and DDOS protection [37], [38]. More broadly, eBPF has been applied in cloud computing for security [39], [40], network optimization [41], [42], virtualization [43], [44], and monitoring [45], [46], [47], [48].

III. Design

Our vision for ViperProbe diverges from comparative techniques [15], [13], [12] which capture *all* metrics, and rather, focuses on determining the set of CriticalMetrics offline coupled with online adaptation. With the changes outline in Section-II-C we argue the collection of all metrics is un-scalable, unnecessary, and can be improved upon.

We eschew the blackbox approach to metric collection, and instead moved to a *graybox* approach informed by offline CriticalMetrics identification coupled with online techniques and dynamic configuration. Specifically, we aim to develop instrumentation which enables precise, uniform control of metrics *per-container* or *per-service*. Then, using knowledge about predefined and standardized design patterns inherent to microservices, we tailor our metrics collection to eliminate costly metrics and thus minimize overheads.

Thus, the challenges for realizing ViperProbe are developing:

1) Algorithms and tools for offline analysis to determine the CriticalMetrics
2) A scalable metric collection framework for instrumenting offline analysis and online dynamic changes

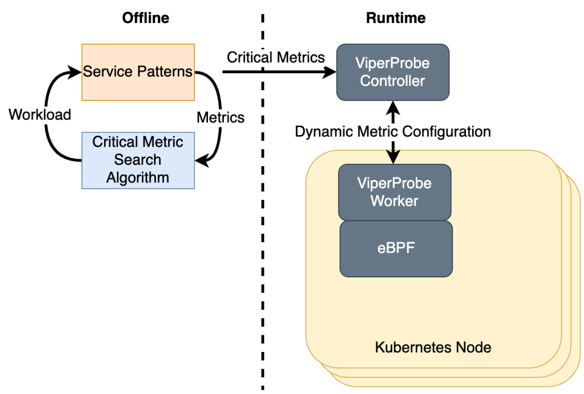

Figure 5. ViperProbe

A. CriticalMetrics

As outlined in Section-II-C, the explosion in heterogeneity, scale, and complexity of metrics makes the collection of all metrics untenable. Intuitively, by reducing the breadth of metrics and instead moving towards depth the associated performance of management tasks also improve (Section-V-A). Our discussion in Section-V-A further motivates, however, not all metrics share equal performance cost. As such, the goal of CriticalMetrics identification is to balance achieving the precision needed to optimally performance manage while finding the minimal cost metrics.

There are two key challenges for identifying the CriticalMetrics:

1) A search algorithm for identifying metrics
2) An offline framework for enabling this search algorithm

1) Identifying CriticalMetrics: Intuitively, metrics tied to the specific critical path of each service are more useful for managing those services. Thus, we argue that CriticalMetrics generally are low-level metrics and require search algorithms to discover the more precise, effective metrics.

These algorithms for identifying the CriticalMetrics can be naive (e.g., brute-force or domain-specific), statistical (clustering [13] or correlations [12]), or based on machine learning (e.g., DeepLearning [15]). Other techniques [49], [50] use metrics provided by the application, or framework (e.g., Kubernetes, OpenShift, Istio) as their performance indicators. We leave a thorough treatment of appropriate algorithms for future work. In our current prototype described in Section IV, we use k-shape [51] clustering to perform offline analysis to determine the CriticalMetrics.

2) Offline Analysis Environment: The purpose of the offline framework is to allow specific microservice patterns to be tested and analyzed in isolation. In particular, this testing framework should support both representative workloads and microservices while allowing fast and efficient testing of different scenarios. The framework also must support a variety of measurements since different performance tasks (e.g., scheduling, debugging) have unique goals. Although the framework can be an emulator or simulator, these often lack fidelity. Alternately, testing in production can introduce effects for end users [9]. When running in production, testing frameworks can introduce performance abnormalities and this ”observer effect” needs to be controlled [9]. In our preliminary prototype, we test using a replica of full production networks. Replicas avoid the challenges of simulators or testing in production, but can be expensive and cumbersome to replicate large production clusters. For ViperProbe, we replicate production for example microservice deployments.

B. Dynamic Metrics Implementation

While dynamics and adaptiveness are at the core of most recent efforts to enable efficient and scalable observability, in practice, many of the existing [12], [13], [15] efforts treat metrics as blackboxes, containing millions of different metrics. We find that this mismatch occurs because most monitoring tools lack fine-grained control over sampling rate, metric collection procedure, or behavior. Instead these alternative tools focus on techniques for interpreting the deluge of metrics. We rethink this approach, beginning with an examination of contemporary tools for kernel-level monitoring.

Existing kernel-level monitoring fall into one of three classes: the first, e.g., Strace [52], are programmatic but limited to a subset of functionality, e.g., counts, or a subset of systems calls. The second class, e.g., Ftrace [53], [54], are not programmatic or dynamic. Due to their limitations, the first two classes are thus not applicable for designing ViperProbe. The third class are both programmable and dynamic, e.g., Dtrace [55] and eBPF and thus more suitable for our goals.

We build ViperProbe on eBFP because of its expressiveness, depth, configurability and of its broader support than Dtrace. The dynamicity and configurability we envision for ViperProbe are not inherent to eBPF given its security and runtime constraints.

C. ViperProbe As a Part of Larger Work

We compare ViperProbe with similar microservice performance analysis tools. We argue ViperProbe builds on these works in a few capacities.

Only ViperProbe is built with the intention for dynamic tuning of metrics. Other works like MicroRCA [49], Loud [50],

		System Data		Application Data		CriticalMetric Identification		
Tool	Goal	Implementation	Dynamic	Implementation	Dynamic	Initial Metrics	Algorithm	Search Period
ViperProbe	Instrument CriticalMetrics	eBPF	Yes	eBPF (uprobes)	Yes	Subset	Unspecified	Offline Initially + Online Adjustment
Sieve [13]	Identify CriticalMetrics	OS Performance Counters	No	Sysdig	No	Subset	Clustering	Offline
Seer [15]	Root Cause for SLA Violations	OS Performance Counters	No	Distributed Tracing	No	All	Deep learning	Offline Training
Pythia (Vision) [12]	Root Cause Analysis	Unspecified	Yes	OSProfiler	Yes	High-level Traces	Statistical Correlation	Online
MicroRCA [49]	Root Service Analysis	Kubernetes & Istio	No	Kubernetes & Istio	No	All	Personalized Pagerank	Offline Training
Loud [50]	Faulty Service Localization	iostat, sar, vmstat, free, ps, ping	No	NMPv2c (Application Specific)	No	All	Graph Centrality	Offline Training

Table I
VIPERPROBE COMPARISON WITH RELATED TOOLS

and Seer [15] start with a fixed set of Key Performance Indicators (KPI) and analyze on that static selection. While tools like Pythia [12] and Sieve [13] attempt to identify the CriticalMetrics online or offline, neither perform both. With ViperProbe we propose a new technique for offline analysis, but at the same time, design the tool to enable dynamic online adjustment. In this paper, we do not explore ViperProbe for online analysis and instead focus on initial offline analysis. However, we believe that ViperProbe's support for both offline and online analysis is unique and an area for future work. Lastly, we believe that ViperProbe's eBPF metrics provide deeper visibility than classic performance counters, and are consistent across applications. Many of the tools in Table-I use application specific metrics or traces [13], [15], [50], while ViperProbe's implementation is agnostic to the application. We believe ViperProbe is the first distributed eBPF-based metric tool aimed at microservices.

IV. Prototype

Our prototype of ViperProbe was built in python using IOVisor's [56] BCC tools for instrumenting eBPF probes. We use gRPC [57] for communication between the Controller and Worker nodes, and then use Kafka as our data collection agent. For our storage and visualization of the metric data, we used Postgres [58] and Grafana [59].

ViperProbe is tightly coupled with Kubernetes, relying on the Kubernetes API for node discovery and pod deployment information. The ViperProbe Controller loads a configuration YAML provided offline by developers. The ViperProbe Controller then sets a *watch* on the Kubernetes API for the pod resource to track new, relocated, or terminated pods, updating Worker configurations accordingly.

V. Preliminary Evaluation

Experimental Setup: All our experiments are performed on Amazon EC2 with 1 master of 8 vCPU and 16Gb of memory and 5 nodes of 8 vCPUs and 16Gb of memory. We deploy Google's microservice Hipster Shop [60], using Locust [61] to simulate load of 1800 users.

ViperProbe Overhead For this experiment, we examined the performance of ViperProbe for collecting finer-grained metrics and for sampling these metrics. We record the median latency and CPU overheads in Figure-6. From this figure, we observe that there is significant benefits in focusing on critical metrics, for example, a system that only requires the cache metric incurs half the overhead of a system which requires the Runq latency metric.

Surprisingly, the figure indicates that sampling has a minimal effect in reducing overhead. These results motivate future work optimizing eBPF in high-velocity paths, including the possibility of support for native sampling.

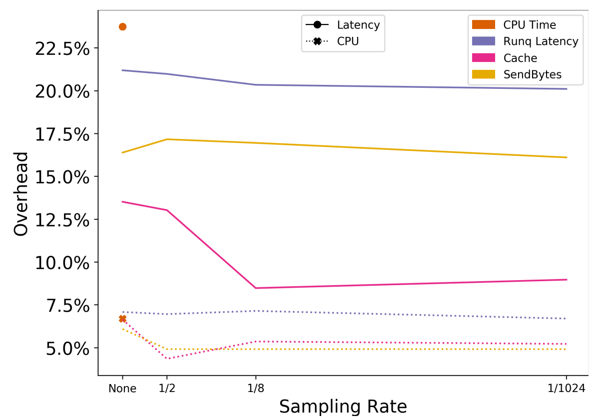

Figure 6. Sampling Median Overheads (Solid lines show Response Latency, Dotted show CPU)

A. Autoscaling

Next, we demonstrate the benefits of the CriticalMetrics by applying them to horizontally autoscale services.

To do this, we compare generic Kubernetes autoscaling [62] using 50% CPU and Memory utilization against a specialized version of autoscaling based on our CriticalMetrics. The specialized version sets thresholds on the metrics identified as CriticalMetrics. To identify the CriticalMetrics, we employ k-Shape [51] clustering for each service coupled with offline analysis. We list the identified CriticalMetrics in Table-III.

In Table- II we present the results of autoscaling, we observe that ViperProbe results in fewer replicas in all services except the recommendation services. For the recommendation service, we observe that ViperProbe allocates over 200% more pods.

In analyzing the servicemesh application, we observe that the recommendation service is a critical bottleneck which is used by many other services (recommendations appear on every page served). Thus, this is the service that should be scaled and not the others (e.g., FrontEnd or Currency) which are being over scaled by Kubernetes.

To illustrate this point, in Figure 7, we explore the number of HTTP500 errors which arise when a request fails due to a lack of resources. In particular, we focus on the request types that leverage the Recommendation service. We note that ViperProbe's specialized metrics allows us to significantly reduce the number of errors. We anticipate that with more fine-tuned system, i.e., better thresholding, we can further reduce these errors.

In these experiments, fine-grained, tailored metrics from ViperProbe better predicted service failure and identified crucial bottlenecks thus enabling preemptive scaling of the appropriate services such as the Recommendation Service.

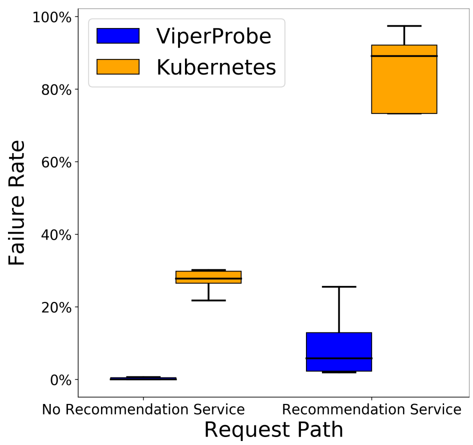

Figure 7. Autoscaling Request Failure Rates

Deployment	ViperProbe	Kubernetes
FrontEnd Service	4	20
Recommendation Service	13	5
Currency Service	6	20
Cart Service	1	1

Table II
SCALE SIZES

Container	Metrics
Redis	userfaults, runq
Currency Service	userfaults, runq
Currency Proxy	runq, sentbytes
Cart Service	userfaults, runq
Cart Proxy	runq, sentbytes
Recommendation Service	userfaults, runq
Recommendation Proxy	runq, sentbytes
FrontEnd Service	runq, sentbytes
FrontEnd Proxy	runq, sentbytes

Table III
CRITICALMETRICS

VI. DISCUSSION

General Observability: Our work takes the first step towards using "patterns" to inform and guide the use of metrics in diagnosis tasks. We believe that similar insights can be used for distributed tracing [31], [32], [35], [33], [34] where "design patterns" can help to localize and improve techniques that explore or analyze traces. Included in this observation is our stress for the need for stronger unity among metrics, tracing, and logs in servicemesh systems.

Broader Offline Analysis: In addition to code-oblivious approaches to analyzing "patterns", we envision that programming languages can be applied to the patterns. In fact, recent work [63] has demonstrated success in using language techniques to analyze software that make up several of the operator patterns. We are interested also in the inverse where knowledge about the set of events from a pattern can be used to limit the exploration for non-pattern code [64].

Effectiveness of Building on Patterns: Although patterns only constitute a small fraction of the general deployment, their use in critical locations, e.g., sidecars, provide us with a strong anchor point for placing constraints and bounds on the potential behavior of other components. We are interested in further understanding how the characteristics of other patterns uniquely inform their behavior.

Scheduling Observability: Our work with ViperProbe highlighted the challenges for monitoring co-located services independently. We theorize that modern schedulers [65], [66] could incorporate observability criteria when making placement decisions. Provided constraints inherent to eBPF, co-locating services which share little overlapping critical paths, as is often done for service performance, can significantly alter ViperProbe performance. Work in this area has generally scheduled services with similar performance requirements (e.g. CPU or Memory heavy) on separate nodes for performance optimization. We are intrigued to see if work in this area might improve observability performance.

Broader Applicability of ViperProbe: We view ViperProbe as part of a larger body of work aimed at improving the depth, scope and precision of microservices performance-oriented management. For example, ViperProbe can expand the set of metrics used by existing systems [67], [15], [14] for decision making. With some systems [67], ViperProbe will allow for better microservices placement, for others [14], [15], ViperProbe will improve the prediction accuracy of QoS violations and performance degradation. For other systems [12], [13], ViperProbe will expand the set of levels and configurability of metrics being collected. We are eager to explore additional ways that ViperProbe can be used to improve existing microservice management frameworks.

VII. CONCLUSION

This paper presents our work on ViperProbe and servicemesh observability. Our work focuses on developing a more robust, configurable metric engine for microservices. ViperProbe is, to the best of our knowledge, the first and only eBPF-based high performance monitoring tool specifically aimed at the servicemesh. Through our work, we have highlighted both the increased heterogeneity of services and uniformity of the accompanying design patterns included with the servicemesh. Using offline analysis for these services and shared software, ViperProbe produces low-level, highly informative metrics per-service. We developed a general framework for eBPF metrics and implemented a variety for our evaluation. In our evaluation, we explored runtime trade offs for the tool, explained challenges with eBPF performance, and presented use cases for the tool. With the increased adoption and scale of the servicemesh, microservice-oriented tools like ViperProbe offer practical solutions.

REFERENCES

[1] "Docker," https://www.docker.com/.

[2] "Production-grade container orchestration," https://kubernetes.io/.
[3] "Istio," https://istio.io/.
[4] "Linkerd," https://linkerd.io/.
[5] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, and et al., "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 3–18. [Online]. Available: https://doi.org/10.1145/3297858.3304013
[6] T. Currie, "Airbnb's 10 takeaways from moving to microservices," https://thenewstack.io/airbnbs-10-takeaways-moving-microservices/.
[7] J. Cloud. (2013) Decomposing twitter. https://www.infoq.com/presentations/twitter-soa/?utm_source=infoq&utm_medium=slideshare&utm_campaign=slidesharenewyork.
[8] A. Cockcroft. (2016) Evolution of microservices. https://www.slideshare.net/adriancockcroft/evolution-of-microservices-craft-conference.
[9] 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
[10] N. T. Blog. (2014) Introducing atlas: Netflix's primary telemetry platform. https://netflixtechblog.com/introducing-atlas-netflixs-primary-telemetry-platform-bd31f4d8ed9a.
[11] S. Srivatsan. Observability at scale: Building uber's alerting ecosystem. https://eng.uber.com/observability-at-scale/.
[12] E. Ates, L. Sturmann, M. Toslali, O. Krieger, R. Megginson, A. K. Coskun, and R. R. Sambasivan, "An automated, cross-layer instrumentation framework for diagnosing performance problems in distributed applications," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 165–170. [Online]. Available: https://doi.org/10.1145/3357223.3362704
[13] 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
[14] S. Chen, C. Delimitrou, and J. F. Martínez, "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*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 107–120. [Online]. Available: https://doi.org/10.1145/3297858.3304005
[15] 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.
[16] B. Arzani, S. Ciraci, B. T. Loo, A. Schuster, and G. Outhred, "Taking the blame game out of data centers operations with netpoirot," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 440–453. [Online]. Available: https://doi.org/10.1145/2934872.2934884
[17] S. M. F. III. What led amazon to its own microservices architecture. https://thenewstack.io/led-amazon-microservices-architecture/.
[18] G. Kakivaya, L. Xun, R. Hasha, S. B. Ahsan, T. Pfleiger, R. Sinha, A. Gupta, M. Tarta, M. Fussell, V. Modi, M. Mohsin, R. Kong, A. Ahuja, O. Platon, A. Wun, M. Snider, C. Daniel, D. Mastrian, Y. Li, A. Rao, V. Kidambi, R. Wang, A. Ram, S. Shivaprakash, R. Nair, A. Warwick, B. S. Narasimman, M. Lin, J. Chen, A. B. Mhatre, P. Subbarayalu, M. Coskun, and I. Gupta, "Service fabric: A distributed platform for building microservices in the cloud," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3190508.3190546
[19] J. Jackson. Debugging microservices: Lessons from google, facebook, lyft. https://thenewstack.io/debugging-microservices-lessons-from-google-facebook-lyft/.
[20] C. McCaffrey. Distributed sagas: A protocol for coordinating microservices. Twitter. [Online]. Available: https://www.youtube.com/watch?v=0UTOLRTwOX0
[21] M. Ranney. What comes after microservices? Uber. [Online]. Available: https://www.youtube.com/watch?v=UDC3kwkBvkA
[22] S. E. Daily. Facebook release engineering with chuck rossi. Software Engineering Daily. https://softwareengineeringdaily.com/2019/08/27/facebook-release-engineering-with-chuck-rossi/.
[23] C. Rossi. Rapid release at massive scale. Facebook. https://engineering.fb.com/web/rapid-release-at-massive-scale/.
[24] J. Humble, "Continuous delivery sounds great, but will it work here?" *Commun. ACM*, vol. 61, no. 4, p. 34–39, Mar. 2018. [Online]. Available: https://doi.org/10.1145/3173553
[25] "Design patterns for microservices," 2020, https://docs.microsoft.com/en-us/azure/architecture/microservices/design/patterns.
[26] T. Cerny, M. J. Donahoo, and M. Trnka, "Contextual understanding of microservice architecture: Current and future directions," *SIGAPP Appl. Comput. Rev.*, vol. 17, no. 4, p. 29–45, Jan. 2018. [Online]. Available: https://doi.org/10.1145/3183628.3183631
[27] "Envoy proxy - home," https://www.envoyproxy.io/.
[28] "Ambassador edge stack," https://www.getambassador.io/.
[29] B. Burns and D. Oppenheimer, "Design patterns for container-based distributed systems," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association, Jun. 2016. [Online]. Available: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/burns
[30] "Sidecar pattern," 2020. [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar
[31] 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. USA: USENIX Association, 2007, p. 20.
[32] 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
[33] "Zipkin," https://zipkin.io/.
[34] "Jaeger," https://www.jaegertracing.io/.
[35] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," *ACM Trans. Comput. Syst.*, vol. 35, no. 4, Dec. 2018. [Online]. Available: https://doi.org/10.1145/3208104
[36] A. Oliner, A. Ganapathi, and W. Xu, "Advances and challenges in log analysis," *Commun. ACM*, vol. 55, no. 2, p. 55–61, Feb. 2012. [Online]. Available: https://doi.org/10.1145/2076450.2076466
[37] H. Zhou, D. Porter, R. Tierney, and N. Shirokov, "Droplet: Ddos countermeasures powered by bpf + xdp," 2017. [Online]. Available: https://netdevconf.info/2.1/slides/apr6/zhou-netdev-xdp-2017.pdf
[38] N. Shirokov and R. Dasineni, "Open-sourcing katran, a scalable network load balancer," 2018, https://engineering.fb.com/open-source/open-sourcing-katran-a-scalable-network-load-balancer/.
[39] L. Deri, S. Sabella, and S. Mainardi, "Combining system visibility and security using ebpf," in *Proceedings of the Third Italian Conference on Cyber Security, Pisa, Italy, February 13-15, 2019*, ser. CEUR Workshop Proceedings, P. Degano and R. Zunino, Eds., vol. 2315. CEUR-WS.org, 2019. [Online]. Available: http://ceur-ws.org/Vol-2315/paper05.pdf
[40] H. M. Demoulin, I. Pedisich, N. Vasilakis, V. Liu, B. T. Loo, and L. T. X. Phan, "Detecting asymmetric application-layer denial-of-service attacks in-flight with finelame," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 693–708. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/demoulin
[41] M. A. M. Vieira, M. S. Castanho, R. D. G. Pacífico, E. R. S. Santos, E. P. M. C. Júnior, and L. F. M. Vieira, "Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications," *ACM Comput. Surv.*, vol. 53, no. 1, Feb. 2020. [Online]. Available: https://doi.org/10.1145/3371038
[42] P. Chaignon, K. Lazri, J. François, T. Delmas, and O. Festor, "Oko: Extending open vswitch with stateful filters," 03 2018, pp. 1–13.
[43] N. Amit and M. Wei, "The design and implementation of hyperupcalls," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 97–112. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/amit

[44] A. Bijlani and U. Ramachandran, "Extension framework for file systems in user space," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 121–134. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/bijlani
[45] J. Perry. Monitoring service architecture and health with bpf. Flowmill Inc. https://www.youtube.com/watch?v=J2NWvh3lgJI.
[46] L. Deri. Monitoring containerised application environments with ebpf. ntop. https://www.ntop.org/wp-content/uploads/2019/05/InfluxData_Webinar_2019.pdf.
[47] Improving performance and reliability in weave scope with ebpf. weaveworks. https://www.weave.works/blog/improving-performance-reliability-weave-scope-ebpf/.
[48] I. Babrou. Debugging linux issues with ebpf. Cloudflare. https://www.usenix.org/sites/default/files/conference/protected-files/lisa18_slides_babrou.pdf.
[49] L. Wu, J. Tordsson, E. Elmroth, and O. Kao, "MicroRCA: Root Cause Localization of Performance Issues in Microservices," in *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, Budapest, Hungary, Apr. 2020. [Online]. Available: https://hal.inria.fr/hal-02441640
[50] L. Mariani, C. Monni, M. Pezzè, O. Riganelli, and R. Xin, "Localizing faults in cloud systems," 04 2018, pp. 262–273.
[51] J. Paparrizos and L. Gravano, "K-shape: Efficient and accurate clustering of time series," *SIGMOD Rec.*, vol. 45, no. 1, p. 69–76, Jun. 2016. [Online]. Available: https://doi.org/10.1145/2949741.2949758
[52] "Strace - linux syscall tracer," https://strace.io/.
[53] "Ftrace - function tracer," 2008, https://www.kernel.org/doc/Documentation/trace/ftrace.txt.
[54] G. Borello, "System and application monitoring and troubleshooting with sysdig." Washington, D.C.: USENIX Association, 2015.
[55] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic instrumentation of production systems," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '04. USA: USENIX Association, 2004, p. 2.
[56] "Bcc - tools for bpf-based linux," https://github.com/iovisor/bcc.
[57] "grpc," https://grpc.io/.
[58] "Postgres," https://www.postgresql.org/.
[59] "Grafana," https://grafana.com/.
[60] "Cloud-native microservices demo," https://github.com/GoogleCloudPlatform/microservices-demo.
[61] "Locust," https://locust.io/.
[62] "Horizontal pod autoscaler," https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.
[63] C. Lou, P. Huang, and S. Smith, "Understanding, detecting and localizing partial failures in large system software," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 559–574. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/lou
[64] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A NICE way to test openflow applications," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 127–140. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/canini
[65] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2741948.2741964
[66] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, scalable schedulers for large compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 351–364. [Online]. Available: https://doi.org/10.1145/2465351.2465386
[67] A. Sampaio Junior, J. Rubin, I. Beschastnikh, and N. Rosa, "Improving microservice-based applications with runtime placement adaptation," *Journal of Internet Services and Applications*, vol. 10, 12 2019.