ViperProbe: Using eBPF Metrics to Improve Microservice Observability*

Brown University Computer Science Honors Thesis

Joshua Levin
Brown University
jlevin1@cs.brown.edu

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 dynamic sampling and collection of deep, diverse, and precise system metrics. We also contribute 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. We describe this subset of metrics as the CriticalMetrics.

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, provide an analysis of eBPF metric performance, examine Envoy's metric performance profile, and show ViperProbe's eBPF metrics were significantly more effective for horizontal autoscaling.

1 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 [4] and Kubernetes [17] 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 [10] and Linkerd [13]) 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 [24, 39–41, 47].

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 scale the diversity and number of metrics in distributed systems [30, 69]. As such, there has been significant work [26, 70] 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 metric sampling.

In this paper, we present ViperProbe, an alternative approach and platform for determining and instrumenting the CriticalMetrics for servicemesh systems. We build on the observation that while the servicemesh diversifies the underlying service, it also brings significant uniformity across the system. Specifically, many microservice deployments have adopted microservice design patterns (Section 2.2), which are in fact, more static and stable than the underlying services. The static, shared nature of these components makes them more amenable to offline analysis thereby reducing the complexity and overhead of online techniques.

In this work, we take the first step towards this vision by presenting a framework, ViperProbe, an adaptive eBPF

*Primary Reader: Theophilus A. Benson, Second Reader: Rodrigo Fonseca

Honors Thesis, May 2020
2020.

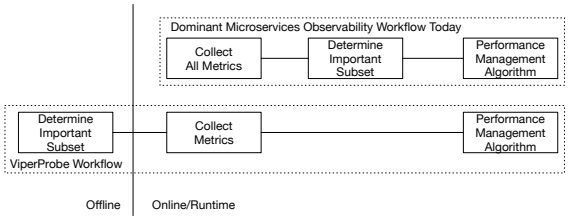

Figure 1: Paradigms for Observability Today

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 [25, 26, 38, 48, 70]. 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. In addition, we use ViperProbe to collect precise performance data which we use to perform statistical analysis of Envoy's performance profile. Our results from Jaccard similarity tests and Kolmogorov–Smirnov tests indicate Envoy sidecars share performance characteristics of *both* the attached service and other Envoy sidecars. Lastly, in our experimental application of ViperProbe for horizontal autoscaling, we found that ViperProbe greatly reduced failure rates (median reduction of 67%, Figure-14) using a subset of CriticalMetrics determined via k-Shape clustering.

2 BACKGROUND

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

2.1 From Microservices to the Servicemesh

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 [53], Microsoft [55], Facebook [54], Twitter [39], Lyft [54, 59], Uber [63], Netflix [40], Airbnb [41]) 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 [42, 47, 52, 64]. 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 [21, 36].

The *servicemesh* describes frameworks [10, 13] and patterns for microservice architectures which provide communication, discovery, security, traffic management, observability, and replication.

2.2 Microservice and Servicemesh Design Patterns

Here, we discuss common microservice design patterns and their use. In Figure 2, we present a canonical servicemesh to illustrate the role of design patterns in modern microservices. The gray boxes are developer code, and the orange boxes are microservice design patterns. These patterns are included

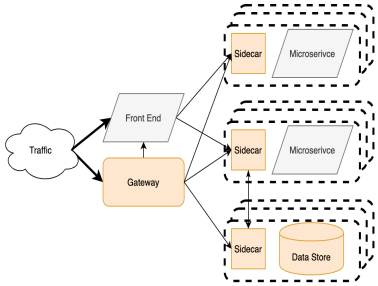

Figure 2: Canonical Microservices deployment

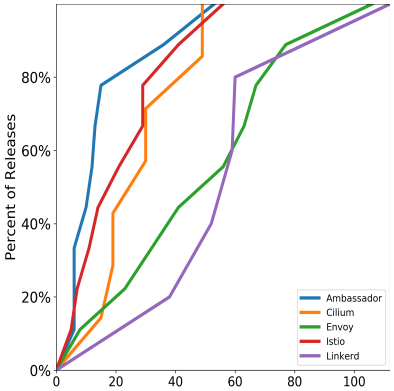

Figure 3: Days between Releases

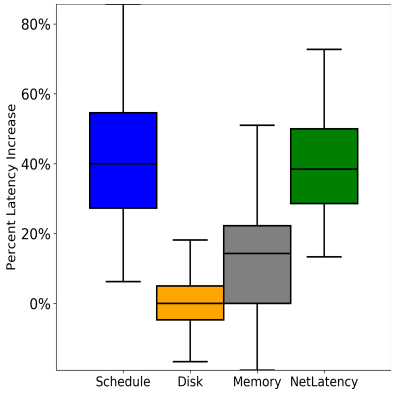

Figure 4: Example Application

in the servicemesh, but can also be deployed separately for traditional microservice applications.

2.2.1 Gateway. Microservices don't utilize Enterprise Service Buses (ESB) as classic monolith architectures did [36]. Instead, often end-users communicate directly with services. 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 [21]. Two popular Gateways are Envoy [5] (configured to be front-edge) and Ambassador [1].

2.2.2 Sidecar/Proxy. The sidecar proxy pattern is a isolated, co-located process that runs alongside each microservice in the servicemesh [32]. The proxy redirects all external network communication through it in order to provide serialization, security, encryption, logging, configuration management, and service discovery [22]. 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. Netflix and Facebook forgo the use of a separate process and instead use a shared library which provides similar benefits and is generally more efficient. The library approach, however, requires developing the shared library for each language/framework in the servicemesh and thus is less general. A popular sidecar proxy, developed by Lyft and used by Istio, is Envoy [5] and is the primary proxy explored in this paper.

2.2.3 Operator. Operators [15] are a recent addition to Kubernetes which enable developers to specify service configurations which are then automatically managed by a Kubernetes operator pod. Microservices are generally stateless and thus traditionally use highly available data stores such as Cassandra, MongoDb, Redis, Postgresql, and MySQL. Operator pods manage custom resource definitions (CRDs) of these data stores in order deploy them as clusters and manage their operational challenges. When deployed on a servicemesh, these data stores often have Envoy or a similar proxy injected by Kubernetes for communication with other services. As these data stores are often unmodified, their performance profiles on the servicemesh are similar to their offline, classic profiles which have preexisting research.

2.2.4 Orchestration. The highly distributed and fragmented design of microservices requires more thorough orchestration and choreography among services to implement features like transactions, overload control, or versioning. Often the orchestration pattern requires both a coordinator and shared API among services, but it also can be implemented solely via shared APIs which is referred to as *choreography*. We share a few examples of orchestrator tools here.

Distributed Sagas: In classic transactional databases, a *Saga* [49] was designed to break long running transactions into smaller ACID compliant transactions. The technique has been adapted by Twitter for its servicemesh to enable ACID-like semantics for servicemesh requests which span multiple independent services [59]. For Distributed Sagas, a central coordinator service splits incoming requests into independent backend requests and applies Two-phase commit like semantics with those requests in order to transact the single request.

Versioning: At Uber, as a result of the high number of dependencies between services and the high-speed, independent development of those services, API consistency between services was challenging. Thus, they built DOSA [63], a shared API, which enforces schema consistency between services by having all services use a shared proxy. Unlike Distributed Sagas, this example does not use a coordinator but rather utilizes shared proxies in order to enforce API consistentcy.

Overload Control: WeChat developed a tool DAGOR [74] to perform load shedding when servers are overloaded. Like DOSA, each microservice is co-located with a DAGOR agent which adds metadata to inbound/outbound requests to communicate with upstream and downstream DAGOR agents as services fail. Using the state information communicated with other agents and shared business logic, DAGOR agents shed lower priority load when servers are under heavy load.

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 [42, 52, 64]. 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 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.

2.3 Observability Challenges

The extreme loose-coupling, scale, and diversity of microservices significantly complicates performance management. As a result, there is a growing body of research on microservice tracing [46, 57, 68], reliance testing [28, 51], and metric/log debugging [26, 48, 70]. At the center of this discussion is the concept of observability.

The core fundamental components of observability in distributed systems are tracing [11, 19, 46, 57, 68], metrics [25, 38, 48, 70], and logging [60]. 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 [30] unique metrics and Uber monitors more than 700 Million [69]. 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 (Figure-5 & Figure-6) 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 [54]. 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 [54].

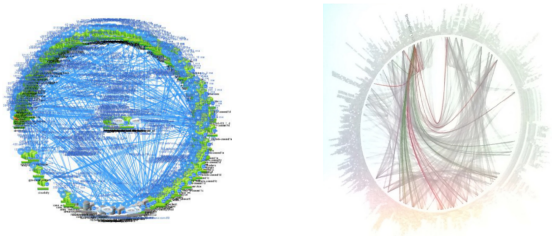

Figure 5: Netflix [40] and Twitter [40] Architectures

Figure 6: Simplified Uber Call Graph [63]

2.4 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 [26, 48, 70], 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-3.1, we further discuss how not all metrics are equally *relevant* to particular services beyond their inherent overhead differences.

2.5 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). Kprobes and tracepoints have existed in the Linux kernel since the early 2000s, however, recent advances made writing more complex programs easier and more practical. eBPF can be compiled by compilers GCC and LLVM from classic C into bytecode which is then injected into the kernel. Generated programs must pass eBPF verification before being loaded. Verification enforces strict constraints such as fixed stack size, reduced instruction set, no floating point arithmetic, and no loops. Programs must be verifiable in *time* and *correctness* as to not crash the kernel or unnecessarily slow the kernel. 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 [67, 75]. More broadly, eBPF has been applied in cloud computing for security [43, 45], network optimization [37, 72], virtualization [23, 29], and monitoring [9, 27, 44, 62].

3 DESIGN

Our vision for ViperProbe diverges from comparative techniques [26, 48, 70] which capture *all* metrics, and rather, focuses on determining the set of CriticalMetrics offline coupled with online adaptation. With the changes outline in Section-2.3 we argue the collection of all metrics is unscalable, 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

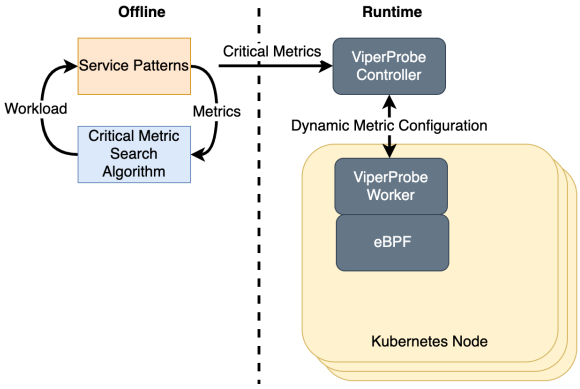

Figure 7: ViperProbe

3.1 CriticalMetrics

As outlined in Section-2.3, 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-5.4.1). Our discussion in Section-2.4 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

3.1.1 Identifying CriticalMetrics. The algorithm for identifying the CriticalMetrics can be naive (e.g., brute-force or domain-specific), statistical (clustering [70] or correlations [26]), or based on machine learning (e.g., DeepLearning [48]). Other techniques [58, 73] 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 4, we use k-shape [61]

		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 [70]	Identify CriticalMetrics	OS Performance Counters	No	Sysdig	No	Subset	Clustering	Offline
Seer [48]	Root Cause for SLA Violations	OS Performance Counters	No	Distributed Tracing	No	All	Deep learning	Offline Training
Pythia (Vision) [26]	Root Cause Analysis	Unspecified	Yes	OSProfiler	Yes	High-level Traces	Statistical Correlation	Online
MicroRCA [73]	Root Service Analysis	Kubernetes & Istio	No	Kubernetes & Istio	No	All	Personalized Pagerank	Offline Training
Loud [58]	Faulty Service Localization	iostat, sar, vmstat, free, ps, ping	No	NMPv2c (Application Specific)	No	All	Graph Centrality	Offline Training

Table 1: ViperProbe Comparison with Related Tools

clustering to perform offline analysis to determine the CriticalMetrics.

Results from Sieve [70] and Pythia [26] demonstrated their tools were most effective when scoping to metrics more specific than traditional CPU utilization or request queue length. It is intuitive that metrics tied to the specific critical path of each service are more useful for managing those services. In our application of ViperProbe in Section-5.4.1 we used runq latency rather than CPU time to measure whether nodes were overworked. Thus, we suggest that CriticalMetrics generally are not high-level and require search algorithms to discover the more precise, effective metrics.

3.1.2 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 [24]. When running in production, testing frameworks can introduce performance abnormalities and this "observer effect" needs to be controlled [24]. 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.

3.2 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 [26, 48, 70] 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.

Strace: [18] is a Linux utility which can be attached to processes via Ptrace in order to provide diagnostic, debugging, and performance insights. Strace is limited to a subset of functionality like counts, and time-spent and natively only supports a subset of system calls.

Ftrace: [20] was an early adapter of Linux kprobes, uprobes, and tracepoints. However, unlike other tracing tools, Ftrace is not programmable, and this severely limits the complexity and logic of metrics you can generate. The Ftrace API is primarily through the commandline, and is generally used for spot-tracing rather than long-lived tracing.

Sysdig: [31] enables developers to apply tcpdump-like semantics for intercepting system calls and delivering the information to userspace. While developers can tune filters, they have limited programmability and are not able to dynamically enable or sample the filters. As of 2018, sysdig shifted to use eBPF to implement its system call redirection process [35].

Dtrace: [34] was created for BSD systems and offers a similar API to eBPF. It is programmable, and dynamic, sharing many of the same tracepoints and depth of eBPF. However, it uses its own D-language, which introduces a learning curve.

eBPF: provides the depth of FTrace with the expressiveness of DTrace while offering a familiar API, the C language. Developers write small C programs which can be attached to kprobes, uprobes, or tracepoints, effectively covering the entire OS stack. eBPF also provides shared maps between user and kernel space which enables real-time bi-directional communication.

In our vision for ViperProbe, our goal is to discard the blackbox approach to monitoring and generating metrics

while providing flexibility, security, and usability. Given these goals, eBPF is the most appropriate of the current metric collection tools to provide depth and configurability. Existing eBPF-based monitoring projects [2] do not directly focus on long-term, adaptive monitoring. The dynamicity and configurability we envision for ViperProbe are not inherent to eBPF given its security and runtime constraints. Our efforts in Section-4 highlight the challenges we found using eBPF while instrumenting ViperProbe.

3.3 ViperProbe As a Part of Larger Work

In Table-1 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 [73], Loud [58], and Seer [48] start with a fixed set of Key Performance Indicators (KPI) and analyze on that static selection. While tools like Pythia [26] and Sieve [70] 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-1 use application specific metrics or traces [48, 58, 70], while ViperProbe's implementation is agnostic to the application. We believe ViperProbe is the first distributed eBPF-based metric tool aimed at microservices.

We leave the identification algorithm for ViperProbe as unspecified and leave that for future work as we use multiple algorithms as part of our evaluation, highlighting ViperProbe's flexibility.

4 IMPLEMENTATION

Our prototype of ViperProbe was built in python using IOVisor's [2] BCC tools for instrumenting eBPF probes. We use gRPC [7] 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 [16] and Grafana [6]. We implemented the metrics outlined in Table-2 for our prototype.

4.1 Orchestration

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.

4.2 eBPF Metrics

Algorithm 1: General eBPF Metric

```
u32 pid_namespace = bpf_get_pdns();
config = get_config_xxx(pid_namespace)
if config and config->enabled and sample(config->rate)
 then
    value = ...
    key = { bpf_log2l(value), pid_namespace }
    stat_histogram.increment(key);
end
```

Through its kprobe, uprobe, and tracepoint functionality, eBPF provides a robust API across the application, kernel, and hardware stacks. However, eBPF has limited program complexity, stack size, and fixed memory allocations, thus making long-term dynamic metric collection challenging. Existing work using eBPF for metrics [2] primarily focuses on commandline tools designed without consideration for system overhead or long-term use. For ViperProbe, we took inspiration from those tools and adapted them to a new model for ViperProbe providing dynamic control, sampling, and long-term health. We show the general workflow for our eBPF programs in Algorithm-1. A central challenge we faced with eBPF was that independent metrics (i.e., cputime and runq latency) commonly share critical functions (i.e., sched_switch). We designed our programs to use *config* parameters, discussed next, which controlled sampling and filter by container.

Area	Metrics	Can Be Sampled
Schedule	cputime (on/off)	No
Schedule	runqueue latency	Yes
Memory	cache ratio	Yes
Memory	kernel page fault	Yes
Memory	user page fault	Yes
Memory	tlb flushes	Yes
Disk	i/o size	Yes
Disk	i/o latency	Yes
Network	send bytes	Yes
Network	recv bytes	Yes
Network	tcp retransmissions	Yes
Network	tcp retransmissions (syn/ack)	Yes
Network	tcp drop	Yes
Dns	dns latency	Yes

Table 2: ViperProbe Metrics

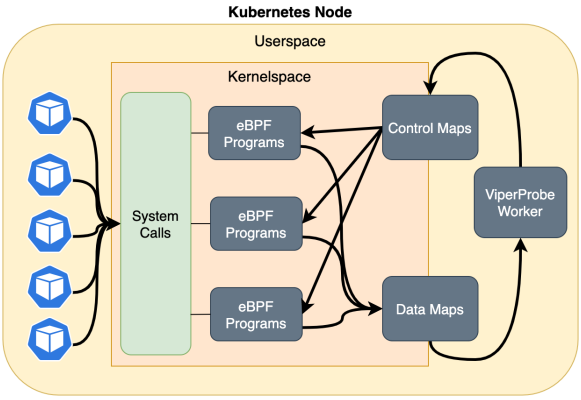

Figure 8: Kubernetes Node with ViperProbe

4.2.1 Container Centric Design. Per the vision of ViperProbe, we attach and attribute metrics *per-container* running on each node (Figure-8). We implement this by tagging kernel events by the current process ID namespace. By default, Docker containers isolate their processes by setting the container PID as the PID namespace. Thus, ViperProbe Workers read out eBPF maps and translate PID namespaces to container ID's using **/proc** before pushing the data to Kafka.

Another challenge we faced with container-centric design was that eBPF programs are attached at the system (or cgroup) level. Thus, a naive implementation of ViperProbe would attach probes purely at the system level, effectively eliminating performance gains of container filtering. Provided we could not attach eBPF programs at finer granularity, our approach was to write *config* information into eBPF maps. On invocation, ViperProbe eBPF programs read a *config*, using the current PID namespace as the key, from shared maps and then either return if disabled, sample if sampled, or run normally (see Algorithm-1). This approach enables selective monitoring per-container, but does incur some additional overhead we discuss in Section-4.3. ViperProbe does not attach probes, however, if no local containers are collecting a metric.

4.3 Sampling

Sampling intuitively is commonly used by observability tools to reduce the volume of information, while preserving the shape [11, 46, 57]. Sampling was the most challenging component of ViperProbe to efficiently implement in eBPF. Here, we present potential implementations, discuss our choice, and then discuss future improvements for eBPF.

Naive: The naive solution for sampling in eBPF is to hard-code sampling configurations into each eBPF program. Under this model, eBPF probes would immediately return without running any logic for a fixed percent of invocations. This solution is static, but is easy to implement and efficient.

Pre-compiled: A potential improvement on the Naive solution is to compile a number of sample-fixed versions of the eBPF programs. To implement this version, ViperProbe would load multiple copies of an eBPF program with different sampling rates hard coded in each version. The first issue with this approach is that ViperProbe would have to attach and detach eBPF probes frequently, potentially degrading performance. Second, this option (and the previous) implements system-wide sampling, and does not support container centric design discussed in Section-4.2.1. This solution is more configurable and just as efficient as the naive option, but is more complex to manage with continuous eBPF program enabling/disabling.

Config Maps: The method which ViperProbe implements (Figure-8) is writing sampling rates to the container configuration maps. Unlike the two previous options, this option requires reading out the *config* for the current PID namespace, but enables both container centric design and sampling. Further, this solution uses a single eBPF function and does not require dynamic attachment, or rewriting of eBPF programs at runtime. Though, this method is less efficient due to the additional cost of looking up the PID namespace in the config maps, but is more flexible.

Dynamic-Compiled: This approach combines the Pre-Compiled and Config Map ideas to dynamically build eBPF programs. In this method, ViperProbe would embed the current *config* map in eBPF programs and re-attach them at runtime. This approach avoids looking up rates in eBPF maps, but adds additional instructions and complexity to the eBPF program itself. It also would result in many more eBPF program attachments and detachments, which can be costly operations. While we believe this would be faster than ViperProbe's Config Map, it adds significant management complexity.

The central challenge which all these approaches attempt to handle is that kprobes, uprobes, or tracepoints are always invoked. This is because kprobes are implemented by inserting a trap into the bytecode of kernel functions. As a result, attaching a kprobe in a high-frequency path like tcp_send or sched_switch can introduce severe overheads (Section-4.4) regardless of the eBPF program speed. These sampling procedures, outlined above, attempt to reduce the impact by making many of these invocations quick. Yet, regardless of these optimizations, attaching probes in high-frequency paths is expensive. Thus, we propose that kprobe/tracepoint native sampling be explored in order to enable efficient eBPF in these high velocity areas.

4.4 eBPF Overheads

Generally, eBPF is regarded a lightweight, low overhead tool for monitoring. While this holds true in industry where

Facebook and Netflix currently run $\tilde{4}0$ and $\tilde{1}0$ eBPF programs per server [50]. Running eBPF programs in the critical path (scheduler, tcp) of services or on high frequency events (page faults, scheduler) can result in considerable Response Latency and CPU Utilization overheads. Our approach to limiting overhead focuses on Sampling (Section- 4.3) and minimizing the number of attached probes. Optimizing ViperProbe's eBPF programs is further discussed as future work in Section-6.

5 EVALUATION

Our evaluation of ViperProbe considers several aspects of the project. First in Section-5.1 we evaluate the overhead introduced by running ViperProbe on a servicemesh. Then, in Section-5.2 we evaluate the costs for particular metrics, and evaluate the effectiveness of ViperProbe's sampling. Our discussion in Section-5.3 uses data from ViperProbe coupled with statistical techniques to provide analysis on the behavior and patterns of Envoy [5] sidecars used by Istio [10]. Last, in Section-5.4 we apply ViperProbe for performance management use cases and compare it to baseline techniques.

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 [3], using Locust [14] to simulate load.

5.1 ViperProbe Overhead

We considered two particular aspects of ViperProbe when evaluating the tool's overhead. First, we looked at CPU utilization while running ViperProbe to collect all metrics with services under load. Second, we evaluated end-to-end latency differences when ViperProbe was running and collecting all metrics. We show the results (Figure-9 & Figure-10) of Hipster Shop running with 1800 users and ViperProbe collecting *all* metrics. We observe that Figure-10 indicates significant latency impact around the 50th percentile, but diminishing, negligible overheads at higher response percentiles. Our explanation for this behavior is that for median response times, ViperProbe probes contribute to slow and impact response times, but for higher percentiles, the relative fixed cost of these probes is dwarfed by external factors which contributed to the response delay. Latency overhead is particularly sensitive to the application, and our experiments found CPU overheads were more consistent. We attribute the modest CPU overhead mostly to our implementation running python, which is highly inefficient, and discuss alternatives in Section-6. In our evaluation of sampling, we saw a minimum of 3-5% CPU overhead across all metrics, which we thus attribute to the system as written python. We believe future work can reduce this significantly by switching to C or C++.

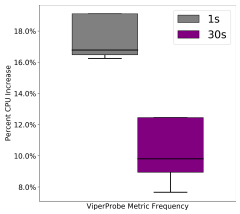

Figure 9: CPU

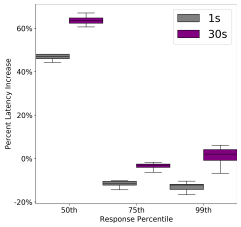

Figure 10: Latency

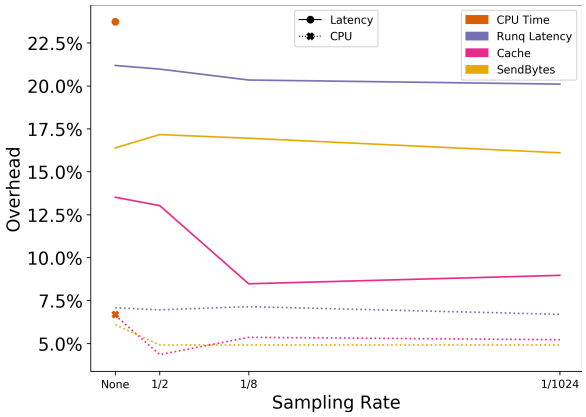

Figure 11: Sampling Median Overheads (Solid lines show Response Latency, Dotted show CPU)

5.2 Sampling Overhead

For this experiment, we examined the performance of ViperProbe sampling, and present finer-grained overheads for a subset of the metrics. Our experiment deployed the Hipster Shop, simulating a load of 1800 users. We record the median latency and CPU overheads in Figure-11. The figures indicate that generally sampling has a minimal effect in reducing overhead. We view that this result confirms our discussion in Section-5.2 on the challenges inherent to eBPF-based sampling. Specifically, these results motivate future work optimizing eBPF in high-velocity paths, including the possibility of support for native sampling.

5.3 The Cost and Performance of Envoy

A prominent contribution of this work is the observation of Microservice Design Patterns (Section-2.2). Here, we focus on the most common pattern, the sidecar proxy pattern. Our evaluation of this pattern begins by exploring the similarity and differences between the performance of services and their Envoy sidecars. Identifying performance similarities

among services and sidecars highlights opportunities where initial offline analysis can generalize across the servicemesh. Consequently, areas of dissimilar performance can be targeted with online techniques.

5.3.1 Jaccard on k-Shape Clusters. In this experiment we performed k-Shape clustering for each service and sidecar deployed on the servicemesh. We clustered 18 unique metrics into 3 clusters and computed the Jaccard similarity between clusters to identify similar clusters among services and sidecars. The k-Shape clustering algorithm clusters metrics by metric shape similarity. By evaluating the similarity of clusters between services and sidecars, we can determine the viability of applying offline analysis among services/sidecars to reduce metric counts similar to Sieve [70]. We present a few observations from the results of the experiment:

(1) The similarity between distinct services was low, unsurprisingly, and thus implying the need for specialization.
(2) While Istio-injected sidecars performed similarly for some metrics (i.e Disk) their behaviors varied among other metrics (i.e. Runq and Network Bytes). Despite providing the same aggregate functionality, the service behavior impacts sidecar behavior further highlighting the need for initial, specific offline analysis, coupled with online monitoring, and dynamic tuning.
(3) A few unexpected services (i.e. Ad + Product, FrontEnd + Shipping) clustered similarly. This clustering indicates these services may share similar sets of CriticalMetrics, although the thresholds and reasons may be dissimilar. We also see this result as an opportunity for future work to possibly colocate services with similar metric profiles.

Metric	**Service-Sidecar** (11 Total)	**Services** (55 Total)	**Envoy** (55 Total)
Runq	1	29	3
On CPU	0	8	0
Off CPU	2	35	21
Sent Bytes	0	5	24
Recv Bytes	0	10	15
Disk I/O Size	0	0	0
Disk I/O Latency	0	0	0

Table 3: K-S Test Results (# *significant* different)

5.3.2 Kolmogorov–Smirnov Tests. Next we looked to identify the similarity of single metrics across the servicemesh. We performed two-sided Kolmogorov–Smirnov [12] tests on the distributions of each metric, comparing services and sidecar. The Kolmogorov–Smirnov test evaluates whether two distributions are drawn from the same reference distribution. In our tests, a *significant* (P-Value < 0.05) result indicates the distributions are **not** drawn from the same distribution. Two sample Runq Latency distributions are show in Figure-12. We present heatmaps of the P-Value tests in Figures-13, and

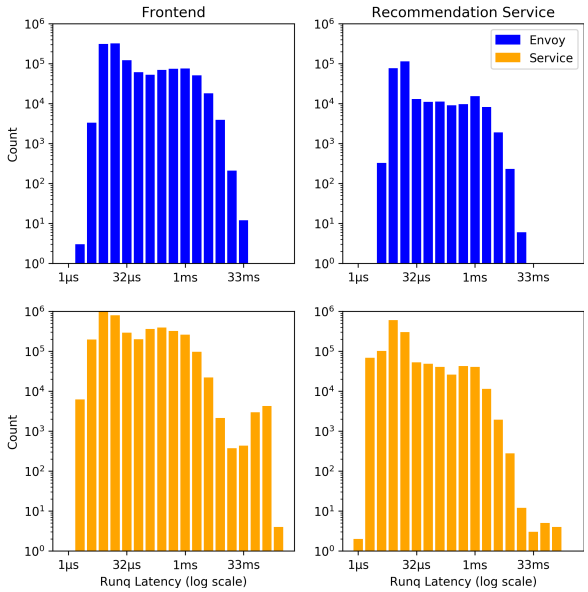

Figure 12: Example Runq Histograms

counts of significant results in Table-3. The heatmaps, specifically the top-left and bottom-right quartiles, help visualize the comparative frequency of significant results between services vs. services and envoy vs. envoy.

From the significant results (Table-3) we present the following conclusions. Along the CPU performance axis (runq, on, off time) service performance profiles diverge in 20-63% of cases. While, Services only diverge in <20% of cases with their co-located sidecars and sidecars only diverge in <38% of comparisons with other sidecars. We view that this apparent contradiction confirms that sidecar performance is the result of *both* the service and Envoy itself. Thus, sidecars share combined characteristics of both their co-located services and other sidecars resulting in lower differences than independent services.

On the Network axis, there as well are consistent patterns. We observe that Services vary less than with CPU performance and that Sidecars vary more. Though, we note there were no instances where sidecars performed differently from their co-located service. This is not entirely surprising to us since sidecars themselves modify and communicate over the network. Thus, this dimension presents as less similar among Envoy instances, though still only 26-44% of sidecar comparisons were significantly different.

The Disk dimensions is unsurprising as sidecars rarely use the disk and our deployment did not include any disk-heavy services. Further, disk events are much less frequent than CPU or Network and thus are tougher to see significantly different distributions.

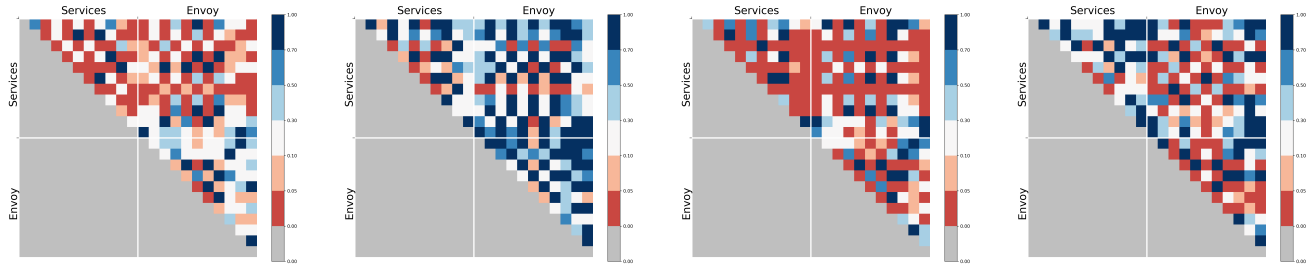

Figure 13: K-S Test P-Value Heatmaps (Grey indicates duplicate or redundant test)

Together, we believe these results reinforce the following conclusions:

(1) Independent services generally have unique performance profiles.
(2) Rarely do services and their associated sidecar have dissimilar performance profiles.
(3) Rarely do sidecar performance profiles vary (though their accompanying services may).

5.4 Use Cases

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

To do this, we compare generic Kubernetes autoscaling [8] 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 [61] clustering for each service coupled with offline analysis. We list the identified CriticalMetrics in Table-5.

In Table- 4 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 14, 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.

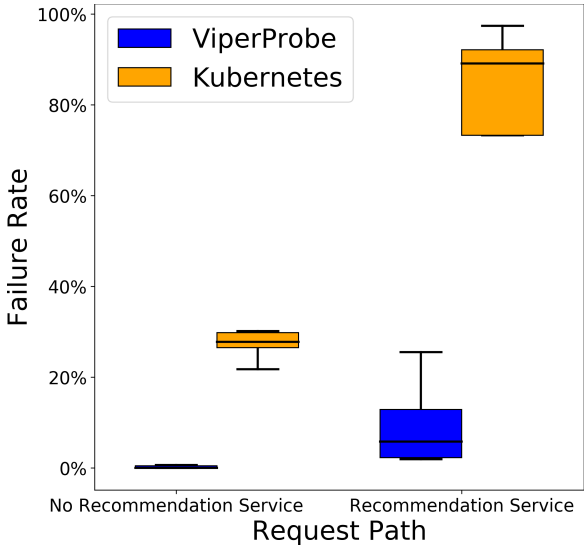

Figure 14: Autoscaling Request Failure Rates

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

Table 4: 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 5: CriticalMetrics

6 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 [11, 19, 46, 57, 68] 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 [56] 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 [33].

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 [66, 71] 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 [38, 48, 65] for decision making. With some systems [65], ViperProbe will allow for better microservices placement, for others [38, 48], ViperProbe will improve the prediction accuracy of QoS violations and performance degradation. For other systems [26, 70], 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.

eBPF super-(sorta)-powers: The growth and popularity of eBPF has prompt many to view the framework as the de-facto future of system monitoring. While we agree eBPF is extraordinarily powerful, we caution this optimism given the challenges that we faced. Given the limitations on how broadly kprobes attach, it is challenging to limit overheads for docker containers. Further, our tests showed sampling inside eBPF programs had limited effect reducing overheads. These challenges make deploying long-term, critical path, eBPF-based monitoring expensive relative to other tools. We believe that future work could explore this juxtaposition between the co-location of services in microservices and the system wide nature of eBPF.

More Efficient System Design: Our current prototype of ViperProbe is based on Python, which introduces its own significant overhead. We plan to explore implementations in other languages such as Go or C++ which consume less resources.

7 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.

8 ACKNOWLEDGEMENTS

First I would like to thank Theophilus A. Benson for his consistent advice, support, and guidance through this process. Without him much of this work would not have been possible. I'd also like to extend thanks to Rodrigo Fonseca for being my second reader through this process.

Additionally I also want acknowledge my many friends in the Computer Science department (too many to list) who listened to me endlessly discuss this project through its many months of work. I'd also like to thank the Brown Computer Science community who was wonderfully welcoming to me. I will forever be grateful to have been able to learn to love Computer Science here as a part of this community.

Lastly, I must extend heartfelt thanks and gratitude to my parents and brother for their consistent support and love throughout this project and my time at Brown.

REFERENCES

[1] [n. d.]. Ambassador Edge Stack. ([n. d.]). https://www.getambassador.io/.

[2] [n. d.]. BCC - Tools for BPF-based Linux. ([n. d.]). https://github.com/iovisor/bcc.

[3] [n. d.]. Cloud-Native Microservices Demo. ([n. d.]). https://github.com/GoogleCloudPlatform/microservices-demo.

[4] [n. d.]. Docker. ([n. d.]). https://www.docker.com/.

[5] [n. d.]. Envoy Proxy - Home. ([n. d.]). https://www.envoyproxy.io/.

[6] [n. d.]. Grafana. ([n. d.]). https://grafana.com/.

[7] [n. d.]. gRPC. ([n. d.]). https://grpc.io/.

[8] [n. d.]. Horizontal Pod Autoscaler. ([n. d.]). https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[9] [n. d.]. Improving performance and reliability in Weave Scope with eBPF. ([n. d.]). https://www.weave.works/blog/improving-performance-reliability-weave-scope-ebpf/.

[10] [n. d.]. Istio. ([n. d.]). https://istio.io/.

[11] [n. d.]. Jaeger. ([n. d.]). https://www.jaegertracing.io/.

[12] [n. d.]. Kolmogorov-Smirnov Goodness-of-Fit Test. ([n. d.]). https://www.itl.nist.gov/div898/handbook/eda/section3/eda35g.htm.

[13] [n. d.]. Linkerd. ([n. d.]). https://linkerd.io/.

[14] [n. d.]. Locust. ([n. d.]). https://locust.io/.

[15] [n. d.]. Operator Hub. ([n. d.]). https://operatorhub.io/.

[16] [n. d.]. Postgres. ([n. d.]). https://www.postgresql.org/.

[17] [n. d.]. Production-Grade Container Orchestration. ([n. d.]). https://kubernetes.io/.

[18] [n. d.]. Strace - linux syscall tracer. ([n. d.]). https://strace.io/.

[19] [n. d.]. Zipkin. ([n. d.]). https://zipkin.io/.

[20] 2008. Ftrace - Function Tracer. (2008). https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[21] 2020. Design patterns for microservices. (2020). https://docs.microsoft.com/en-us/azure/architecture/microservices/design/patterns.

[22] 2020. Sidecar pattern. (2020). https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar

[23] Nadav Amit and Michael Wei. 2018. The Design and Implementation of Hyperupcalls. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 97–112. https://www.usenix.org/conference/atc18/presentation/amit

[24] Dan Ardelean, Amer Diwan, and Chandra Erdman. 2018. Performance Analysis of Cloud Applications. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 405–417. https://www.usenix.org/conference/nsdi18/presentation/ardelean

[25] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. 2016. Taking the Blame Game out of Data Centers Operations with NetPoirot. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 440–453. https://doi.org/10.1145/2934872.2934884

[26] Emre Ates, Lily Sturmann, Mert Toslali, Orran Krieger, Richard Megginson, Ayse K. Coskun, and Raja R. Sambasivan. 2019. An Automated, Cross-Layer Instrumentation Framework for Diagnosing Performance Problems in Distributed Applications. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 165–170. https://doi.org/10.1145/3357223.3362704

[27] Ivan Babrou. [n. d.]. Debugging Linux issues with eBPF. ([n. d.]). https://www.usenix.org/sites/default/files/conference/protected-files/lisa18_slides_babrou.pdf.

[28] Ali Basiri, Lorin Hochstein, Nora Jones, and Haley Tucker. 2019. Automating chaos experiments in production. *CoRR* abs/1905.04648 (2019). arXiv:1905.04648 http://arxiv.org/abs/1905.04648

[29] Ashish Bijlani and Umakishore Ramachandran. 2019. Extension Framework for File Systems in User space. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 121–134. https://www.usenix.org/conference/atc19/presentation/bijlani

[30] Netflix Technology Blog. 2014. Introducing Atlas: Netflix's Primary Telemetry Platform. (2014). https://netflixtechblog.com/introducing-atlas-netflixs-primary-telemetry-platform-bd31f4d8ed9a.

[31] Gianluca Borello. 2015. System and Application Monitoring and Troubleshooting with Sysdig. USENIX Association, Washington, D.C.

[32] Brendan Burns and David Oppenheimer. 2016. Design Patterns for Container-based Distributed Systems. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. USENIX Association, Denver, CO. https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/burns

[33] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. 2012. A NICE Way to Test OpenFlow Applications. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX, San Jose, CA, 127–140. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/canini

[34] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. 2004. Dynamic Instrumentation of Production Systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '04)*. USENIX Association, USA, 2.

[35] Eric Carter. [n. d.]. Introducing container observability with eBPF + Sysdig. ([n. d.]). https://sysdig.com/blog/introducing-container-observability-with-ebpf-and-sysdig/.

[36] Tomas Cerny, Michael J. Donahoo, and Michal Trnka. 2018. Contextual Understanding of Microservice Architecture: Current and Future Directions. *SIGAPP Appl. Comput. Rev.* 17, 4 (Jan. 2018), 29–45. https://doi.org/10.1145/3183628.3183631

[37] Paul Chaignon, Kahina Lazri, Jérôme François, Thibault Delmas, and Olivier Festor. 2018. Oko: Extending Open vSwitch with Stateful Filters. 1–13. https://doi.org/10.1145/3185467.3185496

[38] Shuang Chen, Christina Delimitrou, and José F. Martínez. 2019. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 107–120. https://doi.org/10.1145/3297858.3304005

[39] Jeremy Cloud. 2013. Decomposing Twitter. (2013). https://www.infoq.com/presentations/twitter-soa/?utm_source=infoq&utm_medium=slideshare&utm_campaign=slidesharenewyork.

[40] Adrian Cockcroft. 2016. Evolution of Microservices. (2016). https://www.slideshare.net/adriancockcroft/evolution-of-microservices-craft-conference.

[41] TC Currie. [n. d.]. Airbnb's 10 Takeaways from Moving to Microservices. ([n. d.]). https://thenewstack.io/airbnbs-10-takeaways-moving-microservices/.

[42] Software Engineering Daily. [n. d.]. Facebook Release Engineering with Chuck Rossi. ([n. d.]). https://softwareengineeringdaily.com/2019/08/27/facebook-release-engineering-with-chuck-rossi/.

[43] Henri Maxime Demoulin, Isaac Pedisich, Nikos Vasilakis, Vincent Liu, Boon Thau Loo, and Linh Thi Xuan Phan. 2019. Detecting Asymmetric Application-layer Denial-of-Service Attacks In-Flight with FineLame. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 693–708. https://www.usenix.org/conference/atc19/presentation/demoulin

[44] Luca Deri. [n. d.]. Monitoring Containerised Application Environments with eBPF. ([n. d.]). https://www.ntop.org/wp-content/uploads/2019/05/InfluxData_Webinar_2019.pdf.

[45] Luca Deri, Samuele Sabella, and Simone Mainardi. 2019. Combining System Visibility and Security Using eBPF. In *Proceedings of the Third Italian Conference on Cyber Security, Pisa, Italy, February 13-15, 2019 (CEUR Workshop Proceedings)*, Pierpaolo Degano and Roberto Zunino (Eds.), Vol. 2315. CEUR-WS.org. http://ceur-ws.org/Vol-2315/paper05.pdf

[46] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. 2007. X-Trace: A Pervasive Network Tracing Framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation (NSDI'07)*. USENIX Association, USA, 20.

[47] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, and et al. 2019. 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 (ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 3–18. https://doi.org/10.1145/3297858.3304013

[48] Yu Gan, Yanqi Zhang, Kelvin Hu, Yuan He, Meghna Pancholi, Dailun Cheng, and Christina Delimitrou. 2019. 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)*.

[49] Hector Garcia-Molina and Kenneth Salem. 1987. Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data (SIGMOD '87)*. Association for Computing Machinery, New York, NY, USA, 249–259. https://doi.org/10.1145/38713.38742

[50] Brendan Gregg. [n. d.]. UM2019 Extended BPF: A New Type of Software. ([n. d.]). https://www.slideshare.net/brendangregg/um2019-bpf-a-new-type-of-software.

[51] Victor Heorhiadi, Shriram Rajagopalan, Hani Jamjoom, Michael Reiter, and Vyas Sekar. 2016. Gremlin: Systematic Resilience Testing of Microservices. 57–66. https://doi.org/10.1109/ICDCS.2016.11

[52] Jez Humble. 2018. Continuous Delivery Sounds Great, but Will It Work Here? *Commun. ACM* 61, 4 (March 2018), 34–39. https://doi.org/10.1145/3173553

[53] Scott M. Fulton III. [n. d.]. What Led Amazon to its Own Microservices Architecture. ([n. d.]). https://thenewstack.io/led-amazon-microservices-architecture/.

[54] Joab Jackson. [n. d.]. Debugging Microservices: Lessons from Google, Facebook, Lyft. ([n. d.]). https://thenewstack.io/debugging-microservices-lessons-from-google-facebook-lyft/.

[55] Gopal Kakivaya, Lu Xun, Richard Hasha, Shegufta Bakht Ahsan, Todd Pfleiger, Rishi Sinha, Anurag Gupta, Mihail Tarta, Mark Fussell, Vipul Modi, Mansoor Mohsin, Ray Kong, Anmol Ahuja, Oana Platon, Alex Wun, Matthew Snider, Chacko Daniel, Dan Mastrian, Yang Li, Aprameya Rao, Vaishnav Kidambi, Randy Wang, Abhishek Ram, Sumukh Shivaprakash, Rajeet Nair, Alan Warwick, Bharat S. Narasimman, Meng Lin, Jeffrey Chen, Abhay Balkrishna Mhatre, Preetha Subbarayalu, Mert Coskun, and Indranil Gupta. 2018. Service Fabric: A Distributed Platform for Building Microservices in the Cloud. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article Article 33, 15 pages. https://doi.org/10.1145/3190508.3190546

[56] Chang Lou, Peng Huang, and Scott Smith. 2020. Understanding, Detecting and Localizing Partial Failures in Large System Software. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 559–574. https://www.usenix.org/conference/nsdi20/presentation/lou

[57] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2018. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. *ACM Trans. Comput. Syst.* 35, 4, Article Article 11 (Dec. 2018), 28 pages. https://doi.org/10.1145/3208104

[58] Leonardo Mariani, Cristina Monni, Mauro Pezzè, Oliviero Riganelli, and Rui Xin. 2018. Localizing Faults in Cloud Systems. 262–273. https://doi.org/10.1109/ICST.2018.00034

[59] Caitie McCaffrey. [n. d.]. Distributed Sagas: A Protocol for Coordinating Microservices. ([n. d.]). https://www.youtube.com/watch?v=0UTOLRTwOX0

[60] Adam Oliner, Archana Ganapathi, and Wei Xu. 2012. Advances and Challenges in Log Analysis. *Commun. ACM* 55, 2 (Feb. 2012), 55–61. https://doi.org/10.1145/2076450.2076466

[61] John Paparrizos and Luis Gravano. 2016. K-Shape: Efficient and Accurate Clustering of Time Series. *SIGMOD Rec.* 45, 1 (June 2016), 69–76. https://doi.org/10.1145/2949741.2949758

[62] Jonathan Perry. [n. d.]. Monitoring Service Architecture and Health with BPF. ([n. d.]). https://www.youtube.com/watch?v=J2NWvh3lgJI.

[63] Matt Ranney. [n. d.]. What Comes after Microservices? ([n. d.]). https://www.youtube.com/watch?v=UDC3kwkBvkA

[64] Chuck Rossi. [n. d.]. Rapid release at massive scale. ([n. d.]). https://engineering.fb.com/web/rapid-release-at-massive-scale/.

[65] Adalberto Sampaio Junior, Julia Rubin, Ivan Beschastnikh, and Nelson Rosa. 2019. Improving microservice-based applications with runtime placement adaptation. *Journal of Internet Services and Applications* 10 (12 2019). https://doi.org/10.1186/s13174-019-0104-0

[66] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 351–364. https://doi.org/10.1145/2465351.2465386

[67] Nikita Shirokov and Ranjeeth Dasineni. 2018. Open-sourcing Katran, a scalable network load balancer. (2018). https://engineering.fb.com/open-source/open-sourcing-katran-a-scalable-network-load-balancer/.

[68] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.* Technical Report. Google, Inc. https://research.google.com/archive/papers/dapper-2010-1.pdf

[69] Shreyas Srivatsan. [n. d.]. Observability at Scale: Building Uber's Alerting Ecosystem. ([n. d.]). https://eng.uber.com/observability-at-scale/.

[70] 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 *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware '17)*. Association for Computing Machinery, New York, NY, USA, 14–27. https://doi.org/10.1145/3135974.3135977

[71] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article Article 18, 17 pages. https://doi.org/10.1145/2741948.2741964

[72] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. 2020. Fast Packet Processing with EBPF and XDP: Concepts, Code, Challenges, and Applications. *ACM Comput. Surv.* 53, 1, Article Article 16 (Feb. 2020), 36 pages. https://doi.org/10.1145/3371038

[73] Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. 2020. MicroRCA: Root Cause Localization of Performance Issues in Microservices. In

IEEE/IFIP Network Operations and Management Symposium (NOMS). Budapest, Hungary. https://hal.inria.fr/hal-02441640

[74] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. 2018. Overload Control for Scaling WeChat Microservices. *Proceedings of the ACM Symposium on Cloud Computing - SoCC '18* (2018). https://doi.org/10.1145/3267809.3267823

[75] Huapeng Zhou, Doug Porter, Ryan Tierney, and Nikita Shirokov. 2017. Droplet: DDoS countermeasures powered by BPF + XDP. (2017). https://netdevconf.info/2.1/slides/apr6/zhou-netdev-xdp-2017.pdf