ViperProbe*

Thesis Midterm Report

Joshua Levin
Brown CS '20
jlevin1@cs.brown.edu

ABSTRACT

This paper describes the research and progress of Joshua Levin's thesis project: ViperProbe. ViperProbe is a distributed eBPF-based metrics tool focused on diagnosing performance issues in Microservice and Servicemesh architectures. With many industry companies moving development towards distributed, containerized services using orchestration frameworks such as Kubernetes, frameworks for diagnosing and localizing performance issues require modification as well. ViperProbe, using eBPF, performs deep, highly-specialized metric collection on nodes in a Servicemesh. The use of eBPF allows for more robust metrics to target design patterns of Servicemeshes. This paper presents the state of the project's research and development efforts. It is broken into three sections. First, we provide background on the state of microservices, eBPF, and observability in microservices. Second, we discuss our contributions and design. Third, we discuss our development progress and stage. Fourth, we highlight future directions from our work. Lastly, we describe our next steps in the project.

1 BACKGROUND

As background for the development and design for ViperProbe we summarize recent advances in Microservices, Servicemeshes, Distributed Tracing, and eBPF.

1.1 Microservices

With recent advances in containerization and coordination technologies, many companies such as Netflix [23], Twitter [22], and Airbnb [24] are shifting away from classic monolithic architectures. The shift to *microservices* was largely due to their ability to ease the management and development process for large, highly-distributed, replicated applications. Monolithic architectures have a number of issues [22–24, 27], a few examples include:

(1) Operationally
  (a) Isolating bugs + determining the root of incidents
  (b) Domino crashing and cross-team dependencies
  (c) Independent services cannot be updated alone
  (d) Separate services cannot be scaled independently
(2) Development
  (a) Dependency management across independent services. For example, two services may require different kernel versions or java versions on the same production server.
  (b) Separate teams must coordinate development on a single-application code base

Microservices, using containers and coordination, addresses a number of the above issues by providing [22–24, 27]:

(1) Failure isolation
(2) Dependency isolation and resource constraints
(3) Greater abstraction using API definitions between services
(4) Independent scaling, development, and replication of services

1.1.1 Docker. Docker [5] is the most commonly used containerization framework to run containers. It uses a combination of linux cgroups, namespaces, volumes, and systemd slices to isolate and concurrently run containers on a given host. It is used by Kubernetes [12] to run containers on nodes.

1.1.2 Kubernetes. The primary tool used for coordinating Microservice deployments is Kubernetes (K8s). Kubernetes is a descendant of Borg [34] which was developed by Google and released in 2014. Kubernetes is composed of a master node and worker nodes run in a cluster [16]. The master node orchestrates and schedules *pods* [11] to be run on worker nodes. Pods consist of a collection of co-located docker containers. A *deployment* [15] is a collection of pods that is scheduled to be run. This abstraction allows for replication and scale. The master node runs an API that organizes deployments and *services* [13]. Services route/balance requests to a set of pods that match its *selector*. The selector often corresponds to the microservice function.

1.1.3 Servicemesh. The Servicemesh (Figure- 3) is a design architecture for Microservice applications. Kubernetes introduced the concept of the **Data Plane** and the **Control Plane** (seen in Figure- 1), and the Servicemesh furthers these ideas. In Kubernetes, the Control Plane consists of the Kubernetes master node (running the API) and each worker

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

Thesis Midterm Report, January 2020,
2020.

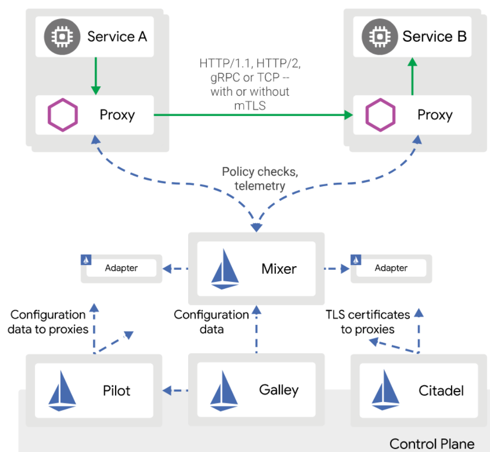

Figure 1: Istio Architecture

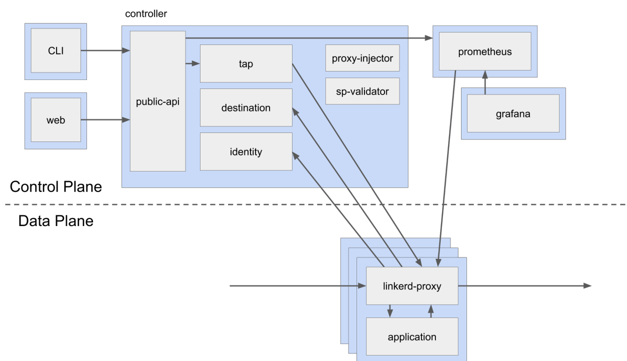

Figure 2: Linkerd Architecture

Figure 3: Service Meshes

node's Kubelet that receives instructions from the master. The Data Plane consists of the Pods running containers on nodes. The Servicemesh model adds to Control Plane by providing more orchestration. It helps developers abstract microservice development and management. With modern microservice applications containing hundreds or thousands of independent services [22–24], these architectures need more structure than what Kubernetes offers alone. The Servicemesh helps address these issues by using proxies, gateways, sidecars, and other standard design patterns which are then controlled by the Control Plane [3]. These proxies are run alongside microservices and allow the Control Plane to manage coordination, communication, security, configuring, and unification.

1.1.4 Istio. Istio [7] (Figure- 1) is one such Servicemesh framework that implements much of this logic for developers and allows for quick development and automated integration with Kubernetes. For the purpose of this project, we choose to focus on Istio + Envoy [6] (the proxy used by Istio), but alternative popular Servicemesh tools exist such as Linkered [10] (Figure- 2). Istio offers a number of pre-built tools to unify communication between services in Servicemeshes. These tools include proxies, api gateways, and ingress/egress gateways [7]. This aspect of Istio relies on Envoy Sidecars that implement much of the logic. These "sidecars" are co-located with sidecars and manage communication/in/out between services.

1.2 Servicemesh Design Patterns

The Servicemesh framework has introduced some common design aspects for microservice applications. These generic helper applications add consistency and uniformity for both the microservice community and individual service developers working in a single Servicemesh. We document some of the most common patterns used today.

1.2.1 Sidecar. The sidecar pattern is a superset of the next two patterns. It describes a shared, isolated, co-located process that runs alongside a microservice [21]. The sidecar could be for inter-service communication (Gateway/Ambassador) but also encapsulate intra-service communication to a database. They have been used for serialization, security, encryption, logging, configuration management, service discovery, or to share logic in an abstract away across multiple services [14].

Gateway. The gateway sidecar serves to protect individual microservices from managing ingress/egress traffic among services. This pattern allows services to centralize request patterns. Examples include aggregating/joining requests, sharing services like TLS, and routing [3]. Gateways may include and add business logic or rules to traffic patterns that are not visible to services. Further, gateways may be run as independent services and not necessarily as sidecars.

Ambassador/API/Proxy. The ambassador or proxy method is more lightweight than the gateway method and abstracts the communication of two services, but does not modify the content of that communication [3]. The major difference between the two is that the ambassador/proxy/api pattern is more tightly integrated with the application. Gateways take ownership of the data, while proxies or APIs provide wrappers for the communication messages. Further, ambassadors/API don't embed any business logic into the requests

as they only pass the information in or out of the application [33].

1.2.2 Orchestrator. The orchestrator design pattern addresses the complications of distributing consistency in a service across multiple microservices. In this pattern an orchestrator service is responsible for coordinating and controlling requests to other services in the mesh. A *Saga* was originally introduced to handle long running database transactions by breaking them into smaller transactions that could be distributed. The idea was expanded into Distributed Sagas [31] where microservice orchestrator services break down consistent requests into smaller independent requests to other services [33].

Another type of orchestrator pattern can help address schema consistency across the servicemesh. Uber built a tool, DOSA [4], for their schema consistency. For Uber, the independent development of thousands of microservices made application schema changes a recurring problem [32]. DOSA works by providing applications API gateways that define schema versions and allow for continuous independent development across teams without schema version issues.

1.3 Observability in Distributed Systems

Distributed tracing and monitoring tools have continued to grow over the past decade, but Microservices present a new challenge for conventional tools. Microservice deployments are highly modular, fragmented, and extremely deep. Thus, classic tracing tools often fail to produce meaningful insights. We highlight some related monitoring projects, describe existing tools and methods, and describe the challenges of microservices.

1.3.1 Related Monitoring Tools. Pythia [19] is a tracing tool for distributed systems, not Microservices, but it's design and structure present relevant context for distributed tracing. Pythia is broken into two planes, much like Servicemesh architectures. The Instrumentation (tracing) Plane implements tracepoints across the entire application stack exposing multiple levels of metrics to the Control Plane [19]. The Control Plane instructs the instrumentation plane on which tracepoints to enable/disable and at what depth of tracing each probe should be operating at. Pythia then learns the expected behavior of applications off basic, sampled metrics. When the Control Plane notices abnormal behavior it automatically enables deeper tracing on area of impact. The work also highlighted how searching the space of instrumentation choices is highly complex, especially as the number of tracing options grows [19].

1.3.2 Microservice Observability. As a result of it's additional complexities, Microservice tracing requires specialized tools such as *Zipkin* [17] and *Jaeger* [9]. Both Zipkin and Jaeger are open-source tracing tools designed to monitor the volume and path of requests into the application. These tools focus on response latency and can help localize at the service-level where potential issues are arising. They work by adding additional information to RPC requests between services and allow users to trace the critical path of incoming requests.

WeChat created it's own internal tool, DAGOR [35], to address *Service Level Object* (SLO) violations across its microservice architecture. A SLO is when a given service fails to handle a request in the agreed upon time. SLOs are more specific, applying to individual services, than *Service Level Agreements* which may define agreements for entire Servicemeshes or Applications. The overload control tool runs on each service and monitors the latency of the request queue. It then procedurally drops low-priority, as define by WeChat, requests that potentially would result in higher-priority violations. The lessons the paper provide two important conclusions for Microservice monitoring: (1) Control (visibility + action) must be decentralized and autonomous in each service (2) Control tools must profile the system *beyond* existing open-metrics and capture application related metrics to assess health

1.3.3 Metrics. Classic infrastructure monitoring tools fail to address the needs of Servicemesh deployments. One such example is the USE [29] (Utilization, Saturation, Error) methodology created by Brendan Gregg. The USE method focuses on system resources and the health of the application from the perspective of the hardware. For microservices, these type of metrics don't necessarily provide meaningful insight for diagnosing issues.

First, individual microservices require different types of metrics relative to their service. Second, these metrics don't necessarily inform the *effectiveness* of the service. Example indicators used commonly for Microservices are the LATEST [25] (Latency, Availability, Throughput, Error Rate, Saturation, Traffic) metrics. These metrics focus on the end-to-end health of customer requests and service. For the most part, customers don't care about hardware and software health or inefficiency if it doesn't impact the above metrics.

A similar approach is the RED [30] (Rate, Errors, Duration) monitoring framework. This framework, like LATEST, focuses on customer-facing and end-to-end health of Microservice architectures. The takeaways from these frameworks are: (1) generic node health and system metrics don't necessarily translate to user-facing health (2) the most important metrics for a servicemesh are metrics that inform and contribute to overall latency and error rate (3) only through a combination of request tracing and can real *observability* be introduced for a servicemesh.

1.4 eBPF for Tracing

Recent advances in the Linux kernel (4.9+, with further improvements at 4.18+) [2] have expanded the use of extended Berkley Packet Filters (eBPF). eBPF is a low-level language for writing programs that are executed in kernel virtual machines. These programs are run through a verifier that strictly validates the programs to ensure they run in fixed-time and will not introduce bugs into the kernel. Despite being a very low language, there are a number of bcc tools [1] for writing the programs in C and attaching them through higher-level languages such as Python and Go.

eBPF is especially effective at providing extremely low-level visibility since it can be attached to both kernel functions and user functions. Existing projects such as Finelame [26], a in-kernel DDOS prevention tool, have shown eBPF's tremendous ability to monitor metrics in real-time and interpret the results. Another group implemented "hyperupcalls" [18] using eBPF to improve the efficiency of nested visualization. Facebook has used eBPF in a tool, Droplet [36], to in-kernel filter out potential attacks or unwanted packets before the packets are allocated space in the kernel. Lastly, another group has used eBPF to build ExtFuse [20] that optimizes fuse mounted filesystems by sharing information between the kernel and user space to avoid unnecessary kernel call.

2 CONTRIBUTIONS

We present ViperProbe as a deep Servicemesh metric collection tool. ViperProbe uses eBPF to collect low-level metrics across every container running on every node in the Servicemesh. True deep observability into Servicemeshes requires logging, tracing, and metric collection. Istio [7] provides general logging and tracing through tools such as jaeger [9] and zipkin [17]. Yet, it only provides top-level metrics such as volume, latency, and error rates [8]. These metrics are often enough to determine *who* or *where* issues are arising, but fail to explain *what* is the cause or *why* a particular service is failing. We seek to expand these metrics to efficiently expand the metrics collected across the Servicemesh in order to help developers identify the root cause of latency or error incidents. We argue these additional metrics and mechanisms provide deeper visibility into potential inefficiencies bottlenecking applications. Given role-specific containers in the Servicemesh as the norm, we argue that classic, generic system-level metrics like cpu utilization, memory utilization, and network i/o inadequately identify the health of individual services. Thus, we propose that with ViperProbe deeply tracing *relevant* resource groups for each application will allow developers to identify inefficient scheduling, design, optimization, resource management, or service scaling.

2.1 New Metrics

As described in section 1.3.3, classic system metrics are inadequate measures for a Servicemesh health as they don't directly contribute to Servicemesh health, generally defined as latency and error rate. Thus, we contribute ViperProbe and it's use of eBPF for metric creation and collection. The use of eBPF enables us to develop more robust metrics tied to application latency and error rate. Specifically, we can measure histograms, per container, metrics like disk i/o latency and frequency, cpu on/off latency, run queue latency (time spent runnable, but waiting on CPU), memory cache hits/misses, and much more. These type of metrics contribute to the speed and ability for services to respond to requests. As Servicemeshes typically reserve similar types of requests to particular containers, we can selectively apply deep monitoring on each application, providing critical insights relevant for each application in the path of a given request. We also submit that eBPF allows us to more robustly develop additional monitors on resources each particular application relies on for responding.

2.2 Design

We present the overall design of ViperProbe in Figure-4. On each node in the mesh we run a logging program that injects eBPF programs and streams data to a kafka server. We run a controller program on a separate server that interacts with the Kubernetes API to get configuration information and control each of the loggers. This cycle of monitoring and updating is demonstrated in Figure-5. Since eBPF runs in the kernel, we only need to inject the metric programs once into the maps and trace as each individual container running on the host invokes kernel functions (Figure-6).

3 DEVELOPMENT

This section describes the progress of the project.

3.1 eBPF Metrics

We have used eBPF to start tracing example metrics including: on-cpu time, off-cpu time, dns-latency, disk i/o size, disk i/o latency, runq latency (time spent runnable but not on cpu), memory cache hit/miss rate, and network byte throughput. Most of these have existing open-source eBPF implementations on the bcc repository we adapted to track per-container. We look forward to tackling more in-depth metrics that can be exposed via eBPF like network latency at each layer in the stack (TCP, IP, NIC). It was critical for our application to attribute these events to specific docker containers. Our expectation was this would be straightforward as eBPF supports the ability to get the current Cgroup a given task is executing under when the eBPF function is invoked. However, most docker containers run in the same Cgroup. They are executed

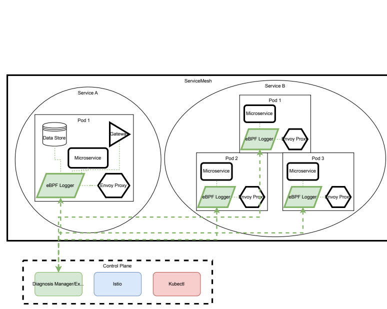

Figure 4: Architecture

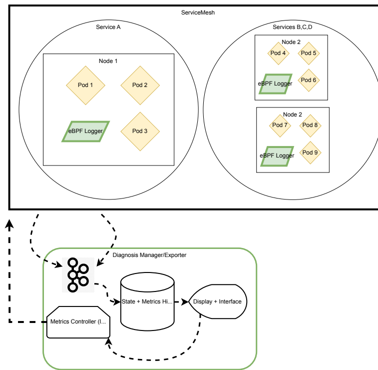

Figure 5: Monitoring System

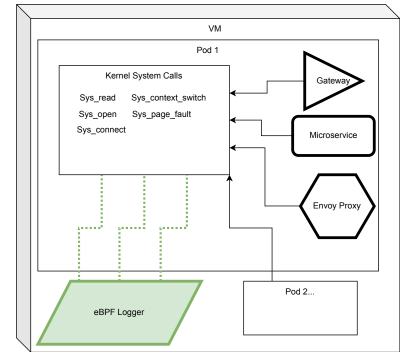

Figure 6: Single Node

in isolated systemd slices. These slices are ephemeral and tough to trace back to a container. The resulting workaround we are using is to get the current PID namesapce. This PID namespace corresponds to the parent process id from the docker container that the process was started under. While docker containers can share PID namespaces, the default is for them not to. We have not yet addressed this assumption, but expect that this would be acceptable for most applications. Thus our current process on each node looks like:

(1) Attach eBPF maps for each metric being collected (maps are keyed by PID namespace)
(2) Listen for configuration from controller node (configuration includes which metrics to track and what pods/containers are running locally)
(3) Poll those maps from python process every X seconds (configurable)
(4) Translate PID namespaces to Docker container IDs
(5) Push messages to Kafka containing metrics and node identity information

The process for now is written in Python. However, we will begin testing overhead soon and may rewrite in Go which has exiting eBPF and docker packages.

3.2 Controller and Kubernetes API

The controller process is also written in Python. This process begins by connecting to the Kubernetes API on the Master node. It collects information on the alive Nodes, Pods, and Services. Then it attempts to push configuration information to every Node, assuming that a logger is running on those nodes. This configuration information tells each node it's identity (which Pods and Services are running on it) and which metrics it should trace. Different nodes running different services may want to trace different metrics. This controller watches events using the kubernetes API and pushes updates to loggers with identity or configuration changes.

The goal is for this process to expose its own API allowing users to specify their own configuration changes or otherwise as applications run. For the moment, we use a CLI that could be exposed straightforwardly as an API in the future.

3.3 Data Pipeline

We chose to use Kafka as the data ingestion engine as it is commonly used in both Zipkin [17] and Jaeger [9] and provides an easy interface for developers to consume the metrics. Each logger pushes data into Kafka. Then our visualization tool consumes off that data stream, joins the data, and produces relevant visualizations/insights. This tool has yet to be built, but is not the focus of the project so it will probably be open source. Further, this design decision enables developers to write their own programmatic consumers for Kafka.

4 NEXT STEPS

These are the following next steps and goals we have for the project:

(1) Write the data consumer to merge metrics and being drawing semantics from the data
(2) Collect information from Servicemesh applications
(3) Investigate more complex/relevant eBPF metrics we can expose
(4) *Quality of life improvements*
    (a) Pod-specific filters (at the moment we do node-level eBPF programs)
    (b) Other aspects to make the controller + loggers more robust and fault tolerant
    (c) Lastly, programmatic logger changes (writing a program that uses statistics to identify in-progress issues and increase monitoring on those nodes)

5 FUTURE WORK

5.1 eBPF

After using eBPF to build ViperProbe we would highlight some issues we encountered and propose some solutions. One such issue was improving the eBPF bcc tools to allow more dynamic control of attaching/detaching eBPF programs. Part of our goal with this project was to be very particular on which metrics we collected to reduce overhead. However, a number of metrics rely on the same kernel functions. An example of this is on/off cpu time and runq latency that both attach to finish_sched_switch. We addressed this by grouping these metrics together in one function and using an all or nothing approach. An alternative approach would be to attach a function with all the logic for all metrics and to read from an eBPF map controlled by the logging process and to use if/else control flow in the eBPF function itself. A possible solution would be to provide eBPF program id control to attach/detach by function ID.

A similar issue we felt with eBPF and docker containers was the challenge in associating processes to docker containers. We assumed process id namespaces were unique to each docker container and relied on that in our tracing. However, this assumption is not necessarily true and breaks our ability to associate metrics with particular containers. Further, it doesn't allow us to run eBPF selectively across containers.

5.2 Dynamic Configuration

We believe that ViperProbe has demonstrated its usefulness as a diagnostic tool. Yet, we believe that its effectiveness in production systems could be increased if it dynamically adjusted collection rates and metrics during incidents. We propose that developing a statistical/machine learning model trained off the data stream that dynamically increased/decreased the sampling/depth of metric collection would improve ViperProbe. eBPF overhead can be significant, especially tracing high-volume events like those in the network stack. Thus, a dynamic process could observe low-intensity metrics or perhaps traces from an external tool like Zipkin/Jaeger and increase the depth of eBPF tracing. We think this addition draws on inspiration from projects like Python [19] and Seer [28] that similarly use machine learning to auto tune metric collection. Further this type of API could be interacted with as Zipkin or Jaeger detect errors that might not be visible to ViperProbe.

REFERENCES

[1] [n. d.]. BCC - Tools for BPF-based Linux. ([n. d.]). https://github.com/iovisor/bcc
[2] [n. d.]. BPF man pages. ([n. d.]). http://man7.org/linux/man-pages/man2/bpf.2.html#VERSIONS
[3] [n. d.]. Design patterns for microservices. ([n. d.]). https://docs.microsoft.com/en-us/azure/architecture/microservices/design/patterns
[4] [n. d.]. Dosa. ([n. d.]). https://github.com/uber-go/dosa
[5] [n. d.]. Enterprise Container Platform. ([n. d.]). https://www.docker.com/
[6] [n. d.]. Envoy Proxy - Home. ([n. d.]). https://www.envoyproxy.io/
[7] [n. d.]. Istio. ([n. d.]). https://istio.io/
[8] [n. d.]. Istio Observability. ([n. d.]). https://istio.io/docs/concepts/observability/
[9] [n. d.]. Jaeger. ([n. d.]). https://www.jaegertracing.io/
[10] [n. d.]. Linkerd. ([n. d.]). https://linkerd.io/
[11] [n. d.]. Pod Overview. ([n. d.]). https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/
[12] [n. d.]. Production-Grade Container Orchestration. ([n. d.]). https://kubernetes.io/
[13] [n. d.]. Service Overview. ([n. d.]). https://kubernetes.io/docs/concepts/services-networking/service/
[14] [n. d.]. Sidecar pattern. ([n. d.]). https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar
[15] [n. d.]. Using kubectl to Create a Deployment. ([n. d.]). https://kubernetes.io/docs/tutorials/kubernetes-basics/deploy-app/deploy-intro/
[16] [n. d.]. Using Minikube to Create a Cluster. ([n. d.]). https://kubernetes.io/docs/tutorials/kubernetes-basics/create-cluster/cluster-intro/
[17] [n. d.]. Zipkin. ([n. d.]). https://zipkin.io/
[18] 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
[19] 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
[20] 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
[21] 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
[22] Jeremy Cloud. 2013. Decomposing Twitter. (2013). https://www.infoq.com/presentations/twitter-soa/?utm_source=infoq&utm_medium=slideshare&utm_campaign=slidesharenewyork
[23] Adrian Cockcroft. 2016. Evolution of Microservices. (2016). https://www.slideshare.net/adriancockcroft/evolution-of-microservices-craft-conference
[24] TC Currie. [n. d.]. Airbnb's 10 Takeaways from Moving to Microservices. ([n. d.]). https://thenewstack.io/airbnbs-10-takeaways-moving-microservices/
[25] Shrenik Dedhia. 2019. A Customer-Focused SLA for a Kubernetes-Based PaaS. (2019). https://static.sched.com/hosted_files/kccncna19/b2/Implementing%20a%20Customer%20Focused%20SLA%20for%20a%20Kubernetes%20Based%20PaaS.pdf
[26] 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

[27] 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

[28] 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)*.

[29] Brendan Gregg. [n. d.]. The USE method. ([n. d.]). http://www.brendangregg.com/usemethod.html

[30] Joab Jackson. [n. d.]. The RED Method: A New Approach to Monitoring Microservices. ([n. d.]). https://thenewstack.io/monitoring-microservices-red-method/

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

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

[33] Madhuka Udantha. [n. d.]. Microservice Architecture and Design Patterns for Microservices. ([n. d.]). https://medium.com/@madhukaudantha/microservice-architecture-and-design-patterns-for-microservices-e0e5013fd58a

[34] 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

[35] 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

[36] 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