Chasing the Microservices Dragon: A View on Microservices Research Challenges

Abstract

There is a shift toward cloud-native applications or microservices-based application deployments. Although there is significant interest and effort within the industry to improve and advance the frontier of microservices systems; however, within the research community, there is a hesitation to explore microservice-based research.

In this paper, we explore cutting edge approaches to managing microservice deployments and identify open research challenges associated with configuration management, service observability, availability, service migration, and service updates. For each challenge, we revisit orthogonal work within the distributed systems community to identify and understand the microservice-specific research challenges. In some instances, we highlight the microservices-specific opportunities that make the problem more tractable within the microservice space than the distributed system space.

1 Introduction

Microservices explore an extreme point in the design of distributed applications, unlike monolithic applications where all the services are packaged together, microservices decompose an application into loosely coupled distributed services. This architectural style is gaining popularity due to its simplicity and modularity, which provides a host of benefits: (i) *faster release cycles* because services can be deployed independently, (ii) *easier management* because each team only has to worry about its own service, (iii) *lower blast radius* during failures because services are loosely coupled, and (iv) *better scalability* because bottleneck components can be individually scaled.

Although there is significant interest in adopting microservices, there is little active research on advancing them. In this survey paper, we take a step back to explore the broad space of microservices and, in turn, highlight novel research challenges. To do this, we compare problems within the microservices domain against traditional distributed system solutions and, in doing so, we identify new challenges. In particular, we focus on the key aspects of microservices that distinguish them from traditional distributed systems—namely, their number, the codebase velocity, and the diversity of components (polyglot-nature).

Current research on microservices, focuses on observability and diagnosis and in understanding the architectural implications of hardware choice on performance. These works demonstrate that some traditional problems, when explored within the microservices context, often lead to new and distinct research challenges. In this work, we explore different dimensions of modern microservices to demonstrate and identify the unique challenges that a microservices deployment adds to the traditional distributed system problems.

- **Migration:** with the growing popularity of microservices, companies are considering re-writing their monolithic applications as micro-services. This process raises several migration-related issues: (i) How does one effectively partition a monolith application into microservices? (ii) How are the different microservice patterns configured to effectively support the newly decomposed application? and (iii) In the case of incremental migration, which parts does one migrate and order?

- **Disruption-free Continous Update:** With the fast-moving nature of microservices, where thousands of updates can take place everyday, techniques need to be developed to ensure disruption free updates with minimal overhead. Pre-existing methods, e.g., continuous rollouts and connection draining, are too slow and resource-intensive for a rapidly changing and evolving codebase.

- **Configuration Management:** Configuration management, optimal configuration selection (i.e., configuration tuning) and debugging misconfiguration, remains a general open issue for distributed system []. This issue is intensified by the velocity and the magnitude of microservices: with microservices, the configuration space is much larger and constantly changing at a faster pace. However, unlike existing approaches, which focus

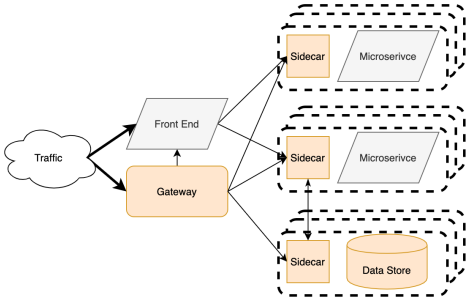

Figure 1: Canonical Microservice Deployment Pattern

on individual applications, the automatic configuration for microservices must explore combinations of services to provide end-to-end improvements.

- **Observability:** The sheer number of components and diversity makes traditional debugging and diagnosis techniques unscalable and impractical. A key issue lies in revisiting the design of monitoring systems to scale and simultaneously tackle underlying data heterogeneity.

Roadmap: In Section 2, we provide background on microservices and highlight the differences between them and traditional monolithic applications. In Section 3, we identify research challenges and conclude with summary remarks in Section 4.

2 Background

In this section, we describe the microservice paradigm (Section 2.1) and the container design patterns which simplify microservice's management (Section 2.2).

2.1 Microservices Paradigm

Microservices differ from traditional distributed applications in that each application is decomposed into a number of services. This loose coupling of services promotes a host of attractive properties: independent code evolution, independent deployment, and independent languages. A consequence of these properties is that, unlike, traditional distributed applications, microservices contain **a large number** of services, each service is **written independently** (with a distinct language and leveraging unique frameworks), and the different services **evolve at different paces**. The implications of these properties is that traditional techniques may be unfit for microservices because **they do not scale** to the number of services, or are **not fast enough to adapt** to code changes, or are **not general enough** to account for a large number of languages and platforms.

Client Interactions: In Figure **??**, we present a canonical microservice deployment. In the canonical tiered deployment, requests come into the network, and a load balancer spreads traffic across the front-end. The front-end interacts with the backend, which instruments business logic and uses a database to persist data.

In the microservice, each incoming client interacts with various services through a gateway pattern, e.g., Kong or Ambassador, which marshalls clients to the appropriate service. The services, in turn, interact with each over a layer seven networking overlay created by a set of reverse proxies: one proxy for each service's container.

Storage: Unlike the monolith scenario, where there is one logical database tier used by all workflows. Within the microservice framework, each service maintains its own database service. While this database-per-service pattern introduces redundancy, it simplifies development because it promotes further loose coupling between teams.

2.2 Design Patterns

The gateway and the ServiceMesh oncepts described above are both examples of microservices design patterns. In this section, we revisit container design patterns and focus on the more popular design patterns which are distinct from the common patterns. Unlike traditional design patterns, which provide a framework for designing and architecting services, the design patterns we describe here are full-fledged codebases that encapsulate specific concepts – here we discuss these core concepts.

Generally, the microservices design pattern provides functionality, which reduces management costs which are naturally higher due to the loose coupling, use of different languages and distributed systems frameworks by different teams. We note that tackling these costs independently at a team level increases the potential for errors, introduces code redundancy and can thus cripple adoption and deployment of microservices.

API Gateway: The gateway unifies and centralizes functionality, e.g., authentication, for managing connections external to the microservice. Emerging gateway patterns provide a broad range of functionality between security and privacy to rate-limiting and throttling to control performance across services.

ServiceMesh: Given the highly distributed nature, there is a need for management frameworks to load balance between replicas, discover replicas, maintain security across distributed components, debug performance problems, capture monitoring data, etc. ServiceMeshes re a framework to explicitly provide these management services in a transparent fashion to free developers from the burden of understanding or tackling issues related to these challenges. A ServiceMesh onsists of two components: the L7-proxy, which acts as the dataplane codebase that encapsulates the various management mechanisms, and a global control plane (or controller), which contains the policies and instructs the data plane on which primitives to use for the different services.

Operator Pattern: These patterns are application-specific,

and they encode operator knowledge required to effectively and efficiently create, configure, and manage instances of the application. To this end, these patterns extend Kubernetes to include a controller for controlling the applications. They are, often, developed to manage stateful services that require detailed knowledge for initialization, scaling, and troubleshooting. Examples include management operators for Redis, Postgres, MySQL, Cassandra.

3 Research Directions

In this section, we explore each of the different issues that modern enterprises face while using a microservices-based infrastructure. In our analysis, we make a few assumptions about their deployment plans, namely that the companies are planning to use cutting edge design patterns, e.g., service mesh, and to employ state of the art management practices, e.g., distributed tracing. Our analysis of the space focuses on availability, performance, and correctness. We leave a more formal discussion of the security and privacy issues to future work.

3.1 Configurations

In addition to configuration correctness, there are several equally important issues: (1) tuning the configurations to provide optimal performance, and (2) debugging misconfiguration problems. This is particularly challenging because the optimal configuration is often a function of the workload, the microservice codebase, and the infrastructure. A change in any of these dimensions may result in a change to the optimal configurations. Thus, with the constant code-velocity of microservices deployment, configuration tuning is particularly challenging. Moreover, the constant code-velocity increases the likelihood of configuration errors [30, 31] — in particular, as code changes, the definition of the correct configuration changes.

State of the Art: The problem of configuration tuning has been explored in the system community within the context of databases [14, 39], infrastructure selection [7], streaming platforms [18] and web services [25, 32, 42]. However, current approaches focus on tuning a specific component of a monolithic application. These techniques fall short for autotuning a microservices deployment because the decomposition of a monolithic requires coordinated tuning of multiple distinct components.

Research Directions: In particular, the concrete research challenges are: to design an autotuning framework for the microservice service mesh. Similar to existing autotuning frameworks, these systems must adapt to changes in the workloads, codebases, and infrastructure. However, unlike other components, this system must holistically configure all the different services within a microservice, which requires novel search algorithms that operate at a different scale than those used in existing systems. Additionally, the fast and quick pace of code changes implies that the search must be fast, e.g., algorithms that operate on the order of hours may be insufficient when code changes occur every hour. We anticipate that other insights about microservices deployments may provide unique opportunities. For example, dependency analysis may help limit the search space.

3.2 Disruption Free Updates

Unlike with monolithic applications, microservice deployments are characterized by their rapid prototyping and deployment cycles. The adoption of continuous integration and continuous delivery developer frameworks (e.g., Jenkins) further fuels these rapid deployment cycles with several large scale organizations claiming multiple code pushes per-hour or per-day to production [19, 36].

State of the Art: For application updates, developers generally set up both versions of the application and gradually shift connections from the older to the newer version by performing a procedure *called draining*. In particular, with draining, new connections are redirected to the newer version and the old version is kept active until all the old requests die out. This approach has two issues: (i) in a resource-constrained environment, having both copies of the application is not feasible. , and (ii) even in the resource-rich environment, i.e., microservice deployed on a cloud, this approach is perfect for short-lived or stateless containers but for long-running and stateful connections, the draining period can be arbitrarily long which prolongs update-times. Note: this problem even arises whether canaries or red-blue deployments are used. The only time these issues do not occur is when the microservice is stateless or stateful with shortlived connections.

Research Directions An open question across all techniques is "how to minimize total draining time across all updates?". In short, a method to effectively support headless migration of connections and their associated state between versions of a service. We believe that this direction can build on two recent efforts: first [34, 35], dynamic software update, which captures developer-written state transformation functions and use these functions to migrate stateful data between the different code versions. The second [21, 22, 33], which breaks the coupling between request processing and state required for request processing by capturing and bundling the associated state and thus allows for migration of this request by transferring the state. While the former approach is general, it does not allow selective migration, whereas the second allows for selective migration but it does not transform the state to tackle application code changes. What we need is a cloud-native library along with a supportive framework that simultaneously supports both state tracking and state transformation in an efficient and performant manner.

3.3 Migration

We analyzed six cases where teams migrated to microservices architecture: Reddit [4], Gojek [3], Spotify [1], Tinder [5], Medium [28] and Mendeley [10]. Across these companies, the main reason for the migration was a lack of flexibility with their current monolith application. In particular, these companies found it hard to effectively extend (adding new features used to break something), manage (developer productivity was decreasing) or deploy (adding a new cluster to the system takes on order of months and huge operational effort) their applications within the current setup. The companies all noted that the loose coupling of microservices deployment could help address their concerns.

In migrating towards a microservice architecture, there are severals steps: (1) partitioning the monolith codebase, (2) replicating and isolating the database to provide a unique database-service for each service, (3) selecting and configuring microservices design patterns (e.g., service-mesh or gateway pattern), (4) creating a build process to support the new microservices deployment workflow.

For a subset of companies, we observed that these companies explored an incremental migration, wherein a few workflows within the monolithic applications are ported to a microservice and others are left within the monolith. To support their incremental deployment, a gateway pattern is used to (1) hide the different deployments from the clients/endusers and (2) support communication between the microservices and the monoliths in transparent manner. For these companies, there are additional challenges. Namely, which order to migrate workflows and how to ensure that the migrated workflows correctly interact with the monolithic application for unmigrated workflows.

State of the Art: There has been significant work on automatically partitioning monolithic applications. These work fall under several categories, the two relevant categories partition an application along the data boundary [11, 26] or partition the application based on communication overhead [13, 20, 40, 41]. However, unlike traditional partitioning, for microservices there is also a need to ensure that services align with team boundaries. Fortunately, we believe that existing partitioning techniques can be extended by introducing team information. In fact, Ignis [40] provides a domain language to help specialize partitioning to cater to such situations. Additionally, existing application partitioning techniques do not replicate/partition the database and rewrite application logic to use this new database service.

Research Directions The open challenges lie in selecting the appropriate design patterns, e.g., service-mesh or gateway patterns, in correctly configuring each of these patterns. While correctly selecting patterns requires an effective search and analysis algorithm, correct configuration requires tools for discovering monolith dependencies [9, 17] and translating these configuration which remain open-challenge. For example, the gateway configuration requires analyzing, parsing, and translating the front-end tier of all the monolith applications.

Companies	Migration Description	Issues
Reddit [4]	Migrated monolith to Kubernetes	Lack of Build System
Gojek [3]	Migrated containerized systems to K8s	
Spotify [1]	Migrated Ingress HTTP Systems to Envoy	-
Tinder [5]	Migrated monolith to Kubernetes + Envoy	Lack of Build System
Medium [28]	Migrated monolith to K8s + Istio	-
Mendeley [10]	Migrated monolith to microservices	

3.4 Dependency Management

The move to a microservice architecture significantly increases the number of components and creates complex dependencies between various services. Many companies claim that individual requests may use hundreds to thousands of services. Given the complex dependencies between services, understanding such complexity is itself an open and crucial challenge.

State of the Art: The state of the art for understanding these dependencies is to use tracing frameworks [2, 6, 29, 37], which captures and externalizes the dependencies. While research techniques exist for analyzing critical paths [16, 27], and quantifying resource requirements, these techniques focus on a request-centric analysis of individual trace graphs. However, multiple requests may be dependent on the same services, and these requests may have different requirements of the service (e.g., because of different code paths). Additionally, while a service may be in the critical path of one request workflow, it may not be in the critical path of other request workflows.

Research Question: To truly understand how to treat a service, we need to take a holistic look across all potential request workflows that include this service and to perform a more holistic analysis of each service's impact across all workflows. This problem is particularly challenging because there are several use cases for such a comprehensive analysis, and each use case may require a different heuristic. The use cases range from performing resource allocation and localizing bottleneck resources to selectively assigning priorities and adaptively assigning deadlines.

3.5 Observability

Observability is often defined as being able to introspect on and to contextualizing an application's internal state from its external outputs. By this definition, monitoring systems are frameworks for capturing and externaling these outputs and they range from logs and metrics to distributed traces.

State of the Art: The current start of the art frameworks focus on capturing and aggregating the monitoring information. A subset of these techniques aim to introspect on the monitoring data by employing deep learning or statistical techniques [12, 15, 38]. However, the broader questions around scalability and cardinality remain open questions. While there is a growing body of work on selectively or adaptively collecting only the traces of interest [8, 23, 24], i.e., traces with

errors or anomalous events, there are no significant tools for adaptively collecting logs and metrics.

Research Direction: To support the growing body of work on using machine learning or statistics to introspect on microservices, we need a new class of monitoring tools. In particular, we need new monitoring tools that scale and deal with microservices-specific issues by adaptively and dynamically collecting traces, logs, metrics rather than simply sampling them. We note that while there is significant effort [15, 38] to address these concerns for tracing, within the space of logs and metrics, the pervalent culture is still to collect everything. We observe that emerging primitives, e.g., eBPF, can be instrumental in this step. A general requirement for these new tools is to be cognizant of the magnitude and quality of the data collected, in particular, the tools should be able to inform the global ML/AI/Statistical framework of the statistical relevance of the input data and the confidence of the values being generated. In essence, these new monitoring building blocks for designing future observability frameworks require a combination of systems and analytical techniques.

4 Conclusion

In this survey paper, we take a step back to analyze problems associated with managing microservices. Our current focus is on migration, performance, availability, and correct. We leave a more thorough treatment of security and privacy to future work. In this work, we identify research challenges related to migration, configuration, observability, debugging/recovery, dependency management, and service updates. Along each dimension, we revisit solutions within traditional distributed system and explore the impact of the microservice paradigm to identify open and unique system challenges.

References

[1] Envoycon 2019.

[2] Jaeger open source, end-to-end distributed tracing. https://www.jaegertracing.io. 2020.

[3] Kubecon cloudnativecon europe 2019.

[4] Kubecon cloudnativecon north america 2018.

[5] Kubecon cloudnativecon north america 2019. https://sched.co/UaVb.

[6] Zipkin. https://zipkin.io.

[7] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 469–482, 2017.

[8] Emre Ates, Lily Sturmann, Mert Toslali, Orran Krieger, Richard Megginson, Ayse K. Coskun, and Raja R. Sambasivan. An automated, cross-layer instrumentation framework for diagnosing performance problems in distributed applications. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 165–170, New York, NY, USA, 2019. Association for Computing Machinery.

[9] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David A. Maltz, and Ming Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. *SIGCOMM Comput. Commun. Rev.*, 37(4):13–24, August 2007.

[10] Mark Boyd. Microservices migration: How elsevier took on its monolith, Jan 2019.

[11] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, page 5, USA, 2004. USENIX Association.

[12] Shuang Chen, Christina Delimitrou, and José F. Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 107–120, New York, NY, USA, 2019. Association for Computing Machinery.

[13] Alvin Cheung, Samuel Madden, Owen Arden, and Andrew C. Myers. Automatic partitioning of database applications. *Proc. VLDB Endow.*, 5(11):1471–1482, July 2012.

[14] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with ituned. *Proceedings of the VLDB Endowment*, 2(1):1246–1257, 2009.

[15] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 19–33, New York, NY, USA, 2019. Association for Computing Machinery.

[16] Xiaofeng Guo, Xin Peng, Hanzhang Wang, Wanxue Li, Huai Jiang, Dan Ding, Tao Xie, and Liangfei Su. Graph-based trace analysis for microservice architecture understanding and problem diagnosis. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page

1387–1397, New York, NY, USA, 2020. Association for Computing Machinery.

[17] Mohammad Hajjat, Xin Sun, Yu-Wei Eric Sung, David Maltz, Sanjay Rao, Kunwadee Sripanidkulchai, and Mohit Tawarmalani. Cloudward bound: Planning for beneficial migration of enterprise applications to the cloud. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 243–254, New York, NY, USA, 2010. Association for Computing Machinery.

[18] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A self-tuning system for big data analytics. In *Cidr*, volume 11, pages 261–272, 2011.

[19] Jez Humble. Continuous delivery sounds great, but will it work here? *Commun. ACM*, 61(4):34–39, March 2018.

[20] Galen C. Hunt and Michael L. Scott. The coign automatic distributed partitioning system. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, page 187–200, USA, 1999. USENIX Association.

[21] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. Stateless network functions: Breaking the tight coupling of state and processing. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 97–112, Boston, MA, March 2017. USENIX Association.

[22] Junaid Khalid and Aditya Akella. Correctness and performance for stateful chained network functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 501–516, Boston, MA, February 2019. USENIX Association.

[23] Pedro Las-Casas, Giorgi Papakerashvili, Vaastav Anand, and Jonathan Mace. Sifter: Scalable sampling for distributed traces, without feature engineering. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 312–324, 2019.

[24] Idit Levine and Mitch Kelley. Kubecon cloudnativecon europe 2019.

[25] Zhao Lucis Li, Chieh-Jan Mike Liang, Wenjia He, Lianjie Zhu, Wenjun Dai, Jin Jiang, and Guangzhong Sun. Metis: Robustly tuning tail latencies of cloud systems. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 981–992, 2018.

[26] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, and et al. Glamdring: Automatic application partitioning for intel sgx. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, page 285–298, USA, 2017. USENIX Association.

[27] Haifeng Liu, Jinjun Zhang, Huasong Shan, Min Li, Yuan Chen, Xiaofeng He, and Xiaowei Li. Jcallgraph: Tracing microservices in very large scale container cloud platforms. In Dilma Da Silva, Qingyang Wang, and Liang-Jie Zhang, editors, *Cloud Computing – CLOUD 2019*, pages 287–302, Cham, 2019. Springer International Publishing.

[28] Xiao Ma. Microservice architecture at medium, Oct 2018.

[29] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 378–393, New York, NY, USA, 2015. ACM.

[30] Ajay Anil Mahimkar, Han Hee Song, Zihui Ge, Aman Shaikh, Jia Wang, Jennifer Yates, Yin Zhang, and Joanne Emmons. Detecting the performance impact of upgrades in large operational networks. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 303–314, New York, NY, USA, 2010. Association for Computing Machinery.

[31] Sonu Mehta, Ranjita Bhagwan, Rahul Kumar, Chetan Bansal, Chandra Maddila, B. Ashok, Sumit Asthana, Christian Bird, and Aditya Kumar. Rex: Preventing bugs and misconfiguration in large services using correlated change analysis. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 435–448, Santa Clara, CA, February 2020. USENIX Association.

[32] Usama Naseer and Theophilus Benson. Configtron: Tackling network diversity with heterogeneous configurations. In *9th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 17)*, 2017.

[33] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Escape capsule: Explicit state is robust and scalable. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, page 10, USA, 2013. USENIX Association.

[34] Karla Saur, Joseph Collard, Nate Foster, Arjun Guha, Laurent Vanbever, and Michael Hicks. Safe and flexible controller upgrades for SDNs. In *Proceedings of the Symposium on SDN Research (SOSR)*, March 2016.

[35] Karla Saur, Tudor Dumitraş, and Michael Hicks. Evolving NoSQL databases without downtime. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, October 2016.

[36] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. Continuous deployment at facebook and oanda. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 21–30. IEEE, 2016.

[37] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[38] Jörg Thalheim, Antonio Rodrigues, Istemi Ekin Akkus, Pramod Bhatotia, Ruichuan Chen, Bimal Viswanath, Lei Jiao, and Christof Fetzer. Sieve: Actionable insights from monitored metrics in distributed systems. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, Middleware '17, page 14–27, New York, NY, USA, 2017. Association for Computing Machinery.

[39] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1009–1024. ACM, 2017.

[40] Nikos Vasilakis, Ben Karel, Yash Palkhiwala, John Sonchack, André DeHon, and James MacGregor Smith. Ignis: scaling distribution-oblivious systems with light-touch distribution. In *PLDI 2019*, 2019.

[41] Fan Yang, Nitin Gupta, Nicholas Gerner, Xin Qi, Alan Demers, Johannes Gehrke, and Jayavel Shanmugasundaram. A unified platform for data driven web applications with automatic client-server partitioning. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, page 341–350, New York, NY, USA, 2007. Association for Computing Machinery.

[42] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 338–350, 2017.