Benchmark Requirements for Microservices Architecture Research

Carlos M. Aderaldo, Nabor C. Mendonça
Programa de Pós-Graduação em Informática
Aplicada, Universidade de Fortaleza
Fortaleza, CE, Brazil
cmendesce@gmail.com, nabor@unifor.br

Claus Pahl
Faculty of Computer Science
Free University of Bozen-Bolzano
Bozen-Bolzano, Italy
cpahl@unibz.it

Pooyan Jamshidi
Institute for Software Research
Carnegie Mellon University
Pittsburgh, PA, USA
pjamshid@cs.cmu.edu

***Abstract*—Microservices have recently emerged as a new architectural style in which distributed applications are broken up into small independently deployable services, each running in its own process and communicating via lightweight mechanisms. However, there is still a lack of repeatable empirical research on the design, development and evaluation of microservices applications. As a first step towards filling this gap, this paper proposes, discusses and illustrates the use of an initial set of requirements that may be useful in selecting a community-owned architecture benchmark to support repeatable microservices research.**

***Keywords*-Microservices; software architecture; research benchmark**

I. Introduction

Microservices have recently emerged as a new architectural style to break up distributed applications into small independently deployable services, each running in its own process and communicating via lightweight mechanisms [1]. These services are built around separate business capabilities and can be written using different programming languages and different data storage technologies. They are usually supported by a fully automated deployment and orchestration machinery, e.g., in the cloud, enabling each service to be deployed often and at arbitrary schedules, with a bare minimum of centralized management [2], often following industry-proven DevOps practices [3].

One of the key aspects of a microservices application is that its complexity is pushed from the components (i.e., services) to the integration (i.e., architectural) level, which requires specific attention from application developers. However, research on microservices has barely started, with an early focus on general architectural principles and automated support [4]. In this respect, there is still a lack of repeatable empirical research on the design, development and evaluation of microservices applications. This is, at least in part, due to the fact that most microservice applications developed so far are proprietary and, therefore, not easily accessible to the software engineering research community.

We believe that having a shared microservices architecture benchmark would provide a unique opportunity for software engineering researchers and practitioners to conduct repeatable empirical studies involving this new architectural style. Such benchmark could be useful, for instance, to investigate the performance impact of new microservices architectures and programming models, and also to compare the effectiveness and limitations of different microservices development methods and automation technologies. Another usage scenario could be to serve as a reference implementation for developers interested in yet unfamiliar with the microservices style [5].

As a first step towards filling this gap, in this paper, we propose and discuss the rationale behind an initial set of requirements that we believe may be useful in selecting a community-owned reference architecture benchmark to support repeatable microservices research. In addition, we illustrate the use of our proposed requirements by using them to guide the selection of possible benchmark candidates amongst existing open source microservices applications.

The rest of the paper is organized as follows. The next section gives an overview of the microservices architectural style and its associated DevOps practices. Section III presents our proposed set of microservices benchmark requirements. Section IV illustrates the use of those requirements in selecting microservices benchmark candidates. Finally, Section V offers some conclusions.

II. Microservices and DevOps

According to Lewis and Fowler, microservices are about functional decomposition often in a domain-driven design context [1]. They are characterized by well-defined and explicitly published interfaces. Each service is fully autonomous and full-stack. Consequently, changing a service implementation has no impact to other services as communication takes place using interfaces only. This achieves loose coupling (usually via REST interfaces) and high cohesion, enabling agility, flexibility and scalability [1].

Sam Newman, in his seminal book [2], defines microservices as small, autonomous services that work together. They are developed focusing on business boundaries, making it obvious where the code lives for a given piece of functionality. And by keeping this focus on an explicit boundary, they help to avoid the temptation for them to grow too large, with all the associated difficulties that this can introduce. All communication between the services themselves is via network calls, to enforce separation between the services and avoid the perils of tight coupling. These services need to be able to change independently of each other and be deployed by themselves without requiring consumers to change [2].

In essence, microservices are an architectural style emerging out of Service-Oriented Architecture (SOA), emphasizing self-management and lightweightness. With respect to the latter, microservices can ideally be packaged, provisioned and orchestrated through the cloud with the usage of lightweight container technology, such as Docker [6], usually following industry-proven DevOps practices [3] to decrease the time between programming a change to a service and transferring that change to the production environment.

In particular, continuous delivery is a DevOps practice that enables on-demand deployment of software to any environment through automated machinery [7]. It is an essential companion of the microservices architectural style as the number of deployable units tends to increase compared to more traditional monolithic architectures [8]. Another critical DevOps practice for microservices applications is continuous monitoring [9], which not only provides developers with performance-related feedback but also facilitates detecting any operational anomalies [10].

III. Requirements for a Microservices Benchmark

The software engineering research community has long relied on open source software systems as common benchmarks with which to conduct and replicate a wide variety of empirical studies. Examples of software engineering benchmarks that have been used in previous studies include the Mosaic web browser [11], the Java Pet Store web application [12], the JHotDraw graphics drawing tool [13], the ArgoUML modeling tool [14], and the Linux [15] and Android [16] operating systems.

As new software development methods and paradigms emerge, new benchmark systems are required in order to support researchers in conducting repeatable empirical studies involving those new software development trends. That is the case of the microservices architectural style, for which there is only a handful of open source systems currently available.

In the following, we describe our proposed initial set of requirements for a microservices research benchmark (see Table I). These requirements reflect how typical microservices applications are currently being developed and delivered into production, as reported by practitioners and industry experts [1], [2], [17]. In particular, they rely on the premise that most microservices applications adopt well-known architectural patterns [18] and follow modern, agile software development practices, such as those advocated by the DevOps [3] and the Twelve-Factor App [19] initiatives. In addition, we have included a few general requirements that we believe would directly benefit the software engineering research community. The last column of Table I presents our rationale for assessing whether (or the extent to which) each requirement would be met by a giving microservices benchmark candidate.

A. Architecture Requirements

These requirements represent desirable attributes of the microservices benchmark from the perspective of its architectural design.

R1: Explicit Topological View: A typical microservices application is composed of several small independently deployable services which may interact asynchronously and indirectly at runtime [2]. These characteristics make it difficult for a developer to fully understand all the services' integration points and responsibilities within the overall application architecture based on its source code alone. A well-documented microservices benchmark should provide an explicit view of its main service elements and their potential communication channels at runtime. Such a view is important to support software engineering researchers in better understanding, exploring and evaluating the benchmark's architectural design decisions and compositional runtime topologies.

R2: Pattern-based Design: The benefits of a pattern-based software architecture, such as ease of maintenance and reuse, have long been recognized by the software engineering community [20]. In this regard, a number of industry-tested architectural patterns have already been proposed to support the creation of scalable and robust microservices applications. Circuit-breaker, API Gateway and Service Discovery are some of the most well-known microservices patterns [2]. The use of such patterns is therefore expected in the design of a microservices benchmark that is representative of how microservices applications are currently being developed and delivered into real-world production environments.

B. DevOps Requirements

These requirements reflect the industry desire that a production-ready microservices application should follow key development practices of the DevOps continuous delivery pipeline [8]. These include easy source code access from a version control repository, and support for typical DevOps practices like continuous integration, automated testing, dependency management, automated deployment, and container orchestration [3]. From a research perspective, meeting these requirements is important in a microservices benchmark, as this would allow software engineering researchers to study, evaluate and experiment with industry-strength DevOps practices and technologies when conducting microservice-based empirical studies.

R3: Easy Access from a Version Control Repository: The use of a Version Control System (VCS) is a crucial aspect of the development of any modern software, even more so in a distributed setting. Using a public distributed VCS such as GitHub or Bitbucket as its main software repository is, therefore, a mandatory requirement for any microservices benchmark candidate, as it allows software engineering researchers and practitioners to have easy access to the benchmark's source code and release history.

R4: Support for Continuous Integration: Continuous integration is a software development practice in which new code created on a developer's machine is automatically integrated with the existing software code base after every code commit submitted to the version control system. Continuous integration tools like Jenkins and TeamCity are responsible

TABLE I
MICROSERVICE BENCHMARK REQUIREMENTS.

Context	Requirements	Assessment Rationale
Architecture	R1: Explicit Topological View	The benchmark should provide an explicit description of its main service elements and their possible runtime topologies.
	R2: Pattern-based Design	The benchmark should be designed based on well-known microservices architectural patterns.
DevOps	R3: Easy Access from a Version Control Repository	The benchmark's software repository should be easily accessible from a public version control system.
	R4: Support for Continuous Integration	The benchmark should provide support for at least one continuous integration tool.
	R5: Support for Automated Testing	The benchmark should provide support for at least one automated test tool.
	R6: Support for Dependency Management	The benchmark should provide support for at least one dependency management tool.
	R7: Support for Reusable Container Images	The benchmark should provide reusable container images for at least one container technology.
	R8: Support for Automated Deployment	The benchmark should provide support for at least one automated deployment tool.
	R9: Support for Container Orchestration	The benchmark should provide support for at least one container orchestration tool.
General	R10: Independence of Automation Technology	The benchmark should provide support for multiple technological alternatives at each automation level of the DevOps pipeline.
	R11: Alternate Versions	The benchmark should provide alternate implementations in terms of programming languages and/or architectural decisions.
	R12: Community Usage & Interest	The benchmark should be easy to use and of interest to its target research community.

for creating a new build of an application and notifying the developer team about the build results. These tools may also trigger the execution of additional tasks, such as code quality check and testing.

R5: Support for Automated Testing: Automated testing tools like Cucumber and Selenium are capable of executing tests, reporting their results and comparing them with earlier test runs. Tests carried out with these tools can be run repeatedly, at any time.

R6: Support for Dependency Management: A dependency management tool like Maven or NPM is responsible for automatically downloading and installing locally all external software artifacts (e.g., components, libraries) required to create a build of a given software product. Those tools provide a specific notation to describe such dependencies, called a *manifest* file. Dependencies specified in a manifest file are usually downloaded from a central software repository.

R7: Support for Reusable Container Images: Microservices applications are typically deployed in a virtualized infrastructure, as that provided by public cloud providers. To accelerate deployment, developers usually rely on lightweight containerized virtualization tools, like Docker, to create reusable container images with the whole software stack and execution environment necessary to run each application component. This allows the application to be easily deployed in the same virtual environment independently of the underlying physical infrastructure (e.g., developer machines, production servers).

R8: Support for Automated Deployment: The deployment configuration of a typical microservice application may vary considerably across different execution environments (e.g., development, staging, production). If those variations were embedded in the application implementation, changing its execution environment would also require changing its source code. This, of course, would make application deployment a very demanding and error-prone task. To alleviate this problem, developers usually specify environment-dependent configuration information in artifacts external to the source code, which are then used by automated deployment tools, such as Chef and Ansible. In general, those tools offer a structured and centralized means to specify the different ways a microservices application should be deployed at runtime.

R9: Support for Container Orchestration: A great feature of containers is that they can be automatically scheduled and orchestrated on top of any physical or virtualized computing infrastructure [6]. Docker Swarm, Kubernetes and Mesos are three of the most commonly used container orchestration tools. These tools provide automated support to address some key challenges of deploying microservices applications, such as service discovery, load balancing, and rolling upgrades [17].

TABLE II
MICROSERVICES BENCHMARK CANDIDATES.

	Technical Characteristics					
				Development Status		
Benchmark Candidate	**Application Domains**	**#Services**	**Languages**	**#Contrib.**	**First Commit**	**Last Commit**
Acme Air	Online store	04	Java, Node.js	05	Jan. 29, 2015	Aug. 25, 2016
Spring Cloud demo apps	Graph processing	08	Java	03	Dec. 15, 2015	Nov. 03, 2016
	Movie recommendation	11	Java	05	May 19, 2015	Nov. 10, 2016
Socks Shop	Online store	19	Java, Go, Node.js	36	Mar. 26, 2016	Jan. 18, 2017
MusicStore	Online store	08	.NET	06	Aug. 08, 2016	Dec. 01, 2016

C. General Requirements

These requirements reflect general benchmark attributes that are not mandatory from a technical perspective, but which would be of great value to the software research community.

R10: Independence of Automation Technology: This requirement prescribes that an ideal microservices benchmark should not be restricted to a single technology with respect to the level of automation associated with the multiple DevOps practices discussed previously. For instance, the benchmark should ideally support the use of multiple tools for automated testing, continuous integration, dependency management, and container orchestration. This would make it easier for researchers to investigate how the use of those different technologies would impact the benchmark's performance and scalability, amongst other quality attributes.

R11: Alternate Versions: Another desirable characteristic for any software architecture benchmark is to provide multiple implementation alternatives. In the particular case of a microservices benchmark, this could mean providing alternate versions of the benchmark's microservices using different programming languages (e.g., Java and Node.js) or different architectural designs (e.g., monolithic vs. decentralized). This would be especially valuable to software engineering researchers interested in comparing different microservice architectures in terms of their design decisions as well as their technological choices.

R12: Community Usage & Interest: The usage history of a research benchmark reflects how often that benchmark has been used by its target research community. In addition, a benchmark that is well documented and easy to customize (e.g., to allow integration with external tools for data collection and analysis) and deploy is likely to further attract the interest of that community. Naturally, selecting a microservices benchmark that is easy to use and has already attracted the interest of other researchers in the field not only increases the confidence on the adequacy of that benchmark for carrying out new research but also promotes repeatability of previous benchmark results.

IV. SELECTING A MICROSERVICE BENCHMARK

To illustrate the benefits of our proposed set of requirements, here we describe how we have used them to guide the selection of possible microservices benchmark candidates from five existing open source microservices demo applications.

A. Benchmark Candidates

We have exhaustively searched the web for existing open source microservices applications that could be good candidates for a microservices research benchmark according to our proposed requirements. As expected, there are only a few publicly-available microservices applications that could satisfy our needs. The five ones that we found most appropriate are described below. Their main characteristics, including application domain, number of containerized services, programming language and development status, are summarized in Table II.

Acme Air: This microservices application simulates the website for a fictitious airline company [21]. It is composed of four separate projects: two server frameworks, developed using Java EE and Node.js, respectively, and two architectural models, following a monolithic architecture and a microservices architecture, respectively.

Spring Cloud demo apps: These are two microservices applications developed to demonstrate some of the fundamental concepts of building microservice-based architectures using Spring Cloud and Docker. The first application [22] implements a graph processing platform for ranking communities of users on Twitter. The second application [23] implements an online movie catalog with an associated movie recommendation service.

Socks Shop: This microservices application simulates the user-facing part of an e-commerce website that sells socks [24]. It was developed using Spring Boot, Go and Node.js, with the specific aim of aiding the demonstration and testing of existing microservice and cloud-native technologies.

MusicStore: MusicStore is a well-known benchmark application which was originally developed by Microsoft to demonstrate ASP.NET features [25]. The team from the Steeltoe framework broke up this application into multiple independent services to illustrate how to use their components together in an ASP.NET core application in a microservices context [26].

B. Results

In order to assess whether each benchmark candidate would satisfy each of our proposed requirements, we have prelimi-

TABLE III
MICROSERVICES BENCHMARK CANDIDATES ASSESSMENT RESULTS.

	Benchmark Assessment Results			
Req.	**Acme Air**	**Spring Cloud demo apps**	**Socks Shop**	**MusicStore**
R1	😐 No explicit view of the overall services topology	☺ Overall services topology clearly described	☺ Overall services topology clearly described	😐 No explicit view of the overall services topology
R2	☺ Circuit Breaker, Service Discovery, Database per Service	☺ Service Discovery, Database per Service, Messaging	☺ Service Discovery, Database per Service, Messaging	☺ Service Discovery, Database per Service
R3	☺ Source code publicly available at GitHub	☺ Source code publicly available at GitHub	☺ Source code publicly available at GitHub	☺ Source code publicly available at GitHub
R4	☹ Build requires multiple steps with no CI support	☺ Uses TravisCI for CI in the Graph Processing App	☺ Uses TravisCI for CI	☹ Build requires multiple steps with no CI support
R5	☹ No automated testing	☺ Uses shell scripts for integration testing	☺ Uses Ruby and Locus, respectively, for integration and load testing	☹ No automated testing
R6	☺ Uses NPM for DM	☺ Uses Maven for DM	☺ Uses Maven and NPM for DM	☺ Uses Nuget for DM
R7	☺ Reusable Docker images	☺ Reusable Docker images	☺ Reusable Docker images	😐 Reusable Docker images only for non .NET business services
R8	☹ No automated deployment	😐 Uses Docker Compose for deployment in a single machine	☺ Supports multiple automated deployment tools (e.g., Docker Swarm, Kubernetes, Mesos)	☹ No automated deployment
R9	☺ Uses IBM Bluemix Container Service for orchestration	☹ No container orchestration	☺ Supports multiple container orchestration tools (e.g., Docker Swarm, Kubernetes, Mesos)	☹ No container orchestration
R10	☹ No support for alternate technologies	☹ No support for alternate technologies	☺ Supports multiple alternate technologies	☹ No support for alternate technologies
R11	☺ Includes both monolithic and microservices versions, in both Java and Node.js	☹ No alternate versions	☹ No alternate versions	☺ Includes both monolithic and microservices versions in .NET
R12	☺ Used as case study in IBM's microservices book [27] and in a recent paper on microservices performance [28]	😐 No previous usage found but recent GitHub activity indicates potential community interest	😐 No previous usage found but recent GitHub activity indicates potential community interest	😐 No previous usage found but recent GitHub activity indicates potential community interest

narily examined each candidate based on their provided documentation and on the information available in their respective software repositories. Our assessment results are summarized in Table III.

As we can see from that table, no single benchmark candidate fully satisfies all our proposed requirements, with Socks Shop being the one which comes close to doing it, fully meeting 10 out of 12 requirements. All the four candidates lack adequate support for one or more key DevOps practices, such as the case with continuous integration, automated testing and container orchestration. This was somewhat expected, however, as those systems were not originally developed targeting a real production environment.

It is worth mentioning that Acme Air is the only candidate that has already been used as case study in the microservices research literature [27], [28]. Another advantage of Acme Air is that it comes in four different versions, covering two radically different architectural designs (monolithic and microservices) and two different implementation technologies (Java and Node.js). Likewise, the MusicStore application is also available in both monolithic and microservices architectural styles. This would make those two applications particularly attractive for software architecture researchers interested in comparing those two architectural designs. On the other hand, both Acme Air and MusicStore lack a clear architectural documentation, which might hinder their early adoption by a wider research community base.

The Spring Cloud and Socks Shop applications, in contrast, are well documented and implemented following a clear design vision based on some well-known DevOps technologies. This makes them relatively easier to understand, use and deploy in the cloud. In addition, the fact that the Spring Cloud graph processing application uses a real-world service (i.e., Twitter), subjected to that service's workload and runtime constraints, means that its execution would require as much planning as with a real production application, which might be seen as a further benefit from the perspective of empirical software engineering researchers.

Finally, we should stress that not all requirements have to be satisfied by a given benchmark candidate for it to be considered useful for empirical research. This decision would largely dependent on the nature of the experimental research to be conducted with the benchmark. For example, the fact that some benchmark candidates are implemented following a single architectural design and using a single implementation technology, such as the two Spring Cloud demo applications, does not preclude them from being used in empirical studies aimed at comparing the effort and performance impact of using different automation technologies.

V. Conclusions

This paper presented an initial set of requirements for a candidate microservices application benchmark to be used in empirical software architecture research. The proposed requirements were discussed and illustrated in the context of selecting possible benchmark candidates amongst five open source microservices applications. Our early results indicate that, although none of the five applications analyzed is mature enough to be used as a community-wide research benchmark, each one of them may already be useful to fulfill the needs and promote the reproducibility of specific empirical studies.

We hope that our proposed requirements can be useful in the process of building a community-wide software architecture research infrastructure, especially as a start point for the discussion of what constitutes an 'ideal' benchmark for conducting empirical microservices research.

Acknowledgements

We thank the workshop reviewers for their constructive comments on how to improve the paper.

References

[1] J. Lewis and M. Fowler, "Microservices," https://martinfowler.com/articles/microservices.html, 2014, [Online; accessed 18-January-2017].

[2] S. Newman, *Building Microservices*. O'Reilly Media, 2015.

[3] I. W. Len Bass and L. Zhu, *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.

[4] C. Pahl and P. Jamshidi, "Microservices: A systematic mapping study," in *Proceedings of the 6th International Conference on Cloud Computing and Services Science*, 2016, pp. 137–146.

[5] P. Winder, "Secure my socks: Exploring microservice security in an open source sock shop," https://goo.gl/yJMbdL, 2016, GOTO Berlin 2016 plenary talk [Online; accessed 19-February-2017].

[6] C. Pahl, "Containerization and the PaaS cloud," *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, 2015.

[7] D. Farley and J. Humble, *Continuous Delivery: Reliable software releases through Build, Test and Deployment Automation*. Addison-Wesley Professional, 2010.

[8] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables DevOps: migration to a cloud-native architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.

[9] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst, "Continuous monitoring of software services: Design and application of the Kieker framework," 2009.

[10] A. Brunnert, A. van Hoorn, F. Willnecker, A. Danciu, W. Hasselbring, C. Heger, N. Herbst, P. Jamshidi, R. Jung, J. von Kistowski *et al.*, "Performance-oriented DevOps: A research agenda," *arXiv preprint arXiv:1508.04752*, 2015.

[11] P. Tonella, R. Fiutem, G. Antoniol, and E. Merlo, "Augmenting pattern-based architectural recovery with flow analysis: Mosaic – a case study," in *Proceedings of the Third Working Conference on Reverse Engineering (WCRE'96)*. IEEE, 1996, pp. 198–207.

[12] A. Mesbah and A. Van Deursen, "Crosscutting concerns in J2EE applications," in *Proceedings of the 7th IEEE International Symposium on Web Site Evolution (WSE'05)*. IEEE, 2005, pp. 14–21.

[13] M. Marin, L. Moonen, and A. van Deursen, "An integrated crosscutting concern migration strategy and its application to JHotDraw," in *Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'07)*. IEEE, 2007, pp. 101–110.

[14] P. Bunyakiati and A. Finkelstein, "The compliance testing of software tools with respect to the UML standards specification – the ArgoUML case study," in *Proceedings of the ICSE Workshop on Automation of Software Test (AST'09)*. IEEE, 2009, pp. 138–143.

[15] A. Israeli and D. G. Feitelson, "The Linux kernel as a case study in software evolution," *Journal of Systems and Software*, vol. 83, no. 3, pp. 485–501, 2010.

[16] M. Asaduzzaman, M. C. Bullock, C. K. Roy, and K. A. Schneider, "Bug introducing changes: A case study with Android," in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR'12)*. IEEE, 2012, pp. 116–119.

[17] K. Bastani, "Building microservices with Spring Cloud and Docker," http://www.kennybastani.com/2015/07/spring-cloud-docker-microservices.html, 2015, [Online; accessed 18-January-2017].

[18] C. Richardson, "A pattern language for microservices," http://microservices.io/patterns, 2014, [Online; accessed 18-January-2017].

[19] A. Wiggins, "The Twelve-Factor App," https://12factor.net, 2012, [Online; accessed 18-January-2017].

[20] F. B. Douglas C. Schmidt, Kevlin Henney, *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*. John Wiley Sons, 2007.

[21] "Acme Air," https://github.com/acmeair/, 2017, [Online; accessed 18-January-2017].

[22] K. Bastani, "Spring Cloud microservice example," https://github.com/kbastani/spring-cloud-microservice-example, 2017, [Online; accessed 18-January-2017].

[23] ——, "Creating a pagerank analytics platform using Spring Boot microservices," http://www.kennybastani.com/2016/01/spring-boot-graph-processing-microservices.html, 2016, [Online; accessed 18-January-2017].

[24] "Socks Shop – a microservices demo application," https://microservices-demo.github.io/, 2016, [Online; accessed 18-January-2017].

[25] "Sample MusicStore application," https://github.com/aspnet/MusicStore, 2017, [Online; accessed 18-January-2017].

[26] "MusicStore – steeltoeoss samples," https://github.com/SteeltoeOSS/Samples/tree/master/MusicStore, 2017, [Online; accessed 18-January-2017].

[27] S. Daya *et al.*, *Microservices from Theory to Practice*. RedBooks, 2015.

[28] T. Ueda, T. Nakaike, and M. Ohara, "Workload characterization for microservices," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, Sept 2016, pp. 1–10.