Runtime models and evolution graphs for the version management of microservice architectures

Yuwei Wang, Denis Conan, Sophie Chabridon, Kavoos Bojnourdi, Jingxuan Ma

Runtime models and evolution graphs for the version management of microservice architectures

Yuwei Wang*†, Denis Conan*, Sophie Chabridon*, Kavoos Bojnourdi†, Jingxuan Ma†
*SAMOVAR, Télécom SudParis, Institut Polytechnique de Paris, France
†EDF R&D, Saclay, France

***Abstract*—Microservice architectures focus on developing modular and independent functional units, which can be automatically deployed, enabling agile DevOps. One major challenge is to manage the rapid evolutionary changes in microservices and perform continuous redeployment without interrupting the application execution. The existing solutions provide limited capacities to help software architects model, plan, and perform version management activities. The architects lack a representation of a microservice architecture with versions tracking. In this paper, we propose runtime models that distinguishes the type model from the instance model, and we build up an evolution graph of configuration snapshots of types and instances to allow the traceability of microservice versions and their deployment. We demonstrate our solution with an illustrative application that involves synchronous (RPC calls) and asynchronous (publish-subscribe) interaction within information systems.**

***Index Terms*—Microservice architecture, version management, model at runtime.**

I. Introduction

In the recent trend to migrate from monolithic architectures to microservice architectures, business logic is split into a set of loosely coupled microservices. Each microservice acts as an independent unit that implements one (small) functionality, and that communicates with other microservices through lightweight communication [1]. The modularity of microservices facilitates their independent replacement and upgradeability [2]. Updating a microservice means bringing into play a new version, configuring new instances to deploy, removing instances of the replaced version, etc. Of course, not all the microservices evolve and are upgraded at the same rate. In addition, every microservice can change at any time and even disrupt the system in some way.

In the ecosystem of enterprises, particularly industrial ones, microservices of an application are usually developed and maintained by different internal teams or subcontractors without a global governance. In addition, the co-existence of legacy systems should be taken into account and the outages of some critical systems can only be once a year. Microservice providers often deploy multiple versions in parallel, offering some specific versions to certain customers or older versions for legacy systems. In this case, the updates of these heterogeneous microservices may go through organisational boundaries, and add dynamics and complexity to reconfiguration deployment. Therefore, it becomes necessary to help software architects model evolving microservices in a uniform manner and take decisions dynamically on version changes while the system is running. Once the decisions of version changes are made by architects, another issue concerns planning and executing the deployment and redeployment of changed microservice applications in an automatic manner. In our approach, we follow the MAPE-K principle of causality between the control loop and the managed system for autonomic computing [3].

The contributions of this paper are twofold. Firstly, we propose runtime models to represent the essential elements of microservice architectures into two views: microservice type model and microservice instance model, respectively describing a structural abstraction of microservice architectures and their specific deployment configurations, which themselves conform to the type model. In addition, these models distinguish synchronous from asynchronous communication. Secondly, we build up an evolution graph: the first part made of configuration type snapshots and the second part of configuration instance snapshots. Every time new or old microservice types are added or removed from the implementation repository, a new node is created and committed in the part of configuration types. Such a node represents the set of software artefacts (microservices, connectors, etc.) that can be used for building the managed system. The second part corresponds to the snapshots of deployed configurations. Every time a decision is made by architects and a change occurs in the managed elements, a new node is created and committed to the part of configurations. Such a node represents the set of deployed entities (deployed instances of microservices, connectors, etc.). Each node of the second part refers to a node of the first graph, i.e. the set of deployed instances conform to the set of available types. Not presented in this paper, but engineered in the prototype, when a new node of the configuration graph is committed, an AI Planner can compute a plan of actions to reconfigure the managed system to obtain the new configuration.

This paper is organised as follows. In section II, we motivate our approach and give the objectives of our work. In Sections III and IV, we describe our main contributions of the runtime model and the evolution graph for microservice architectures by using an illustrative use case and its version management scenarios. In Section V, we review some related works. Finally, we conclude the paper in Section VI.

II. Motivations and objectives

Microservices have advantages in terms of agility and scalability because each microservice becomes an independent

unit of development, deployment, and operation [4]. However, this leads to large numbers of microservices, increasing the runtime management cost. Semantic Versioning (SemVer) [5] is commonly used in software development and microservice versioning to limit the configuration and growth of version numbers [6], [7]. It introduces a set of rules and requirements on how to assign version numbers and whether a new version is backward compatible [8]. Considering the version format of X.Y.Z (Major.Minor.Patch), SemVer informs architects and system administrators, and helps IT teams to anticipate potential breaking changes. We follow this policy to express the versions of all the type elements that need to be versioned in microservice architectures.

Our proposal addresses especially the following typical requirements identified in industrial contexts: (i) In order to rationalise the cost of evolution and maintenance of heterogeneous software solutions, a global and unified view of the ecosystem evolution is necessary; (ii) The continuity of service should be ensured while updating, including the recovery in case of an invalid configuration or any other abnormality; (iii) Evolutionary changes to the system should be traceable.

In fostering automation, self-adaptive systems are able to dynamically adapt their structure and behaviour in a changing environment [3], [9]. In this work, we apply the MAPE-K control loop with microservice versioning. This loop is organised around four activities "Monitor", "Analyze", "Plan", and "Execute" that share a "Knowledge base". More precisely, in this paper, we focus on the knowledge base, and in a secondary manner, the prototype, which is not presented in this paper, automatically plans and executes actions to reconfigure the managed application[1]. Therefore, considering the degree of automation, our proposition is semi-automatic: certain activities of MAPE-K, such as deciding which versions of microservices to use, is done manually by software architects, while the activities of planning and executing are performed automatically.

In order to control and trace changes, we follow the "model at runtime" approach [12]. A runtime model provides a reduced representation of the heterogeneous software elements that are available in the implementation repository and of the managed elements of the running application. This approach has been used in self-adaptive systems to manage complex runtime behaviours [13]. Software adaptations are applied to models maintained at runtime before being executed in the managed applications. In this paper, we propose runtime models as the Knowledge base to abstract and mirror microservice applications. If there are changes occurring in the model, the applications will also change, and vice versa.

To illustrate our contributions, we use a microservice application of a Scientific Research Data Management System as an use case; it is displayed in Figure 1. The main objective of this system is to characterise scientific research data that are produced or used by researchers and engineers during simulation activities. It consists of six microservices. The `Project service` and `File service` manage scientific simulation projects metadata and related research data. The `User service` manages the information on application users and the `Permission service` controls their access permissions. The `Authentication service` verifies the users' login and password via Single Sign-On and the `Logger service` records login history. Each Microservice has its own database and communicates with others through synchronous or asynchronous publish-subscribe interaction modes.

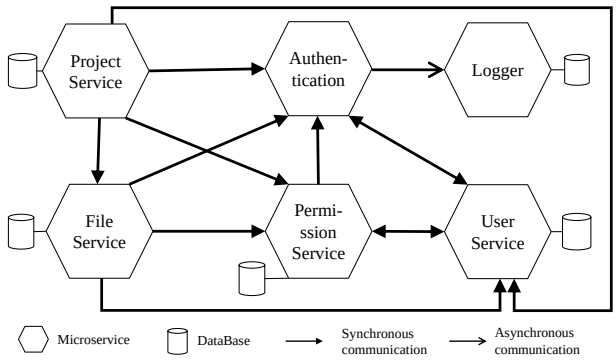

Fig. 1. Illustrative application architecture

III. Runtime Model for Microservice Versioning

In this section, we present the model that we use to abstract a microservice architecture and its versioning. We use model at runtime concepts to provide an unified representation of up-to-date elements in the system driving evolutionary changes. Our proposed model is divided into two parts: the types model (Section III-A) and the instance model (Section III-B). The types model captures the structure of instantiable elements of the evolving microservice applications, and the instance model captures the replicas of corresponding deployable elements of the managed system.

A. Model of types

The model of types is shown in Figure 2. It presents the core elements to specify the configuration types of microservice applications. For reasons of space, the root class of the types, namely ConfigurationType, is not drawn in this diagram. As its name implies, a configuration type aggregates all the types that the configuration [14] of a managed system may contain. These are the types that are available, for example, as code artefacts or as container images in the implementation repositories, i.e. available microservice types, available connector types, and available database system types. Naturally, all the types are uniquely identified by a name and a version: e.g. the `authentication_service` in version `1.0.0` or `rabbitmq` in version `3.9.1`.

The central concept of the model is the microservice type. Each microservice type exposes a set of contract types, which are the "interfaces" provided or required by an instance of the microservice type. These contracts are intermediate entities to

[1]The planning is performed using a PDDL AI planner [10], [11] and the executor uses the Kubernetes API to reconfigure the managed application.

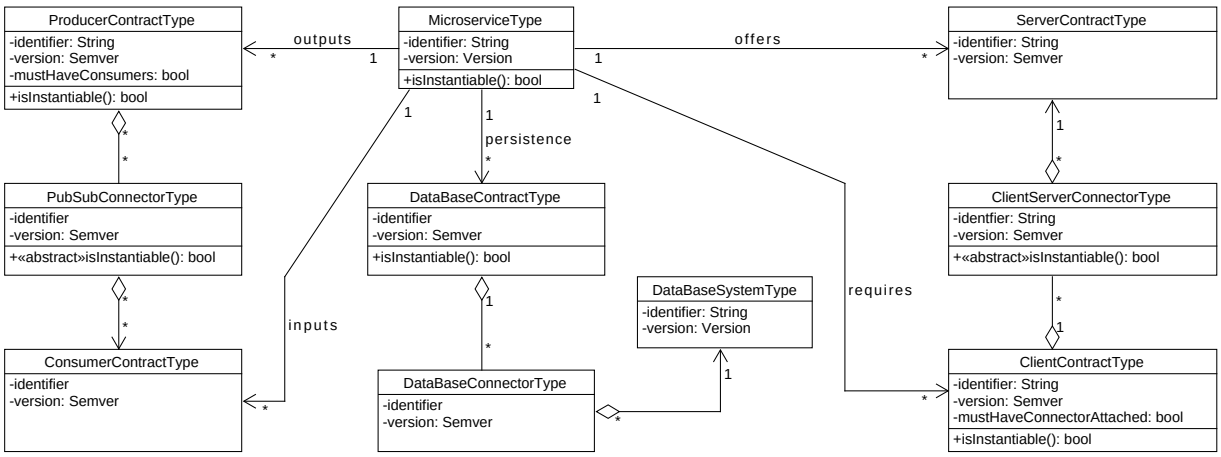

Fig. 2. Model of available types of a configuration type

connectors [15] and are where architects specify the quality of service of the connections, e.g. for producer and consumer contracts [16]. For instance, client-server connectors may be synchronous channels using HTTP-based REST communication and the attribute mustHaveConnectorAttached indicates whether the connection to a server microservice is mandatory, i.e. whether the client may operate in degraded mode without a connection to a compatible server. By definition, considering invariance in static type checking [17], a client contract type is compatible with a server contract type if and only if all the operation declarations of the client contract type are present in the server contract type. Extrapolating from Figure 1, there are six types of microservices. For instance, microservice type `authentication_service` provides contract types to others and requires a server contract type from microservice type `user_service`.

The other two categories of connector types are the database and publish-subscribe connector types. Database contract types are where architects specify the connections to database systems. Every declared database contract types of a microservice type is mandatory. Extrapolating from Figure 1, five of the six microservice types have their own database system type. Publish-subscribe connector types are asynchronous channels. For example, microservice type `authentication_service` publishes its logs that are forwarded to microservice type `logger_service`. Not displayed in Figure 2, we differentiate channel-based and topic-based publish-subscribe systems [18], [19]. Publish-subscribe connectors are typically brokers, e.g. MQTT[2] and AMQP[3] brokers. Microservice applications usually do not involve an enterprise service bus so that producers and consumers must exchange compatible event types [20]. Thus, considering channel-based filtering, a producer contract type is compatible with a set of consumer contract types if and only if the channel names are equal and all the event types produced are accepted by one of the consumer contract types. For topic-based filtering, a producer contract type is compatible with a set of consumer contract types if and only if its topic or routing key (resp. in MQTT and AMQP) matches one of the subscriptions or binding keys (resp. in MQTT and AMQP), and all the event types produced are accepted by the corresponding consumer contract types. In addition, the mustHaveConsumers attribute indicates whether the connection to the producer microservice is mandatory, i.e. whether the producer microservice can operate in degraded mode without compatible consumers.

All of these types of microservices, database systems, contracts and connectors constitute a configuration type of the application. A configuration type is instantiable, namely an instance model can be created that conforms to these types, if and only if (1) client contract types are compatible to attached server contract types, (2) client contract types that must have connectors attached are indeed linked, (3) producer contract types are compatible with connected consumer contract types, (4) producer contract types that must have consumers actually have consumers, and (5) all the database contract types are indeed connected to database systems via database connectors. Any version change of a microservice creates a new microservice type with the same identifier but a changed version. It is up to the software architects to decide which microservice types on which versions can be included in the current configuration type. This is the same for contract types, connector types, and database system types. Finally, every time a change is committed in the type model, the instantiable property is checked.

B. Model of instances

The model of instances is shown in Figure 3. Grey elements come from the model of types. Each instance conforms to a type. A microservice type may have multiple microservice replicas that represent deployed microservice instances. Similarly, a database type may have multiple instances, and a

[2]MQTT: https://mqtt.org/
[3]AMQT: https://www.amqp.org/

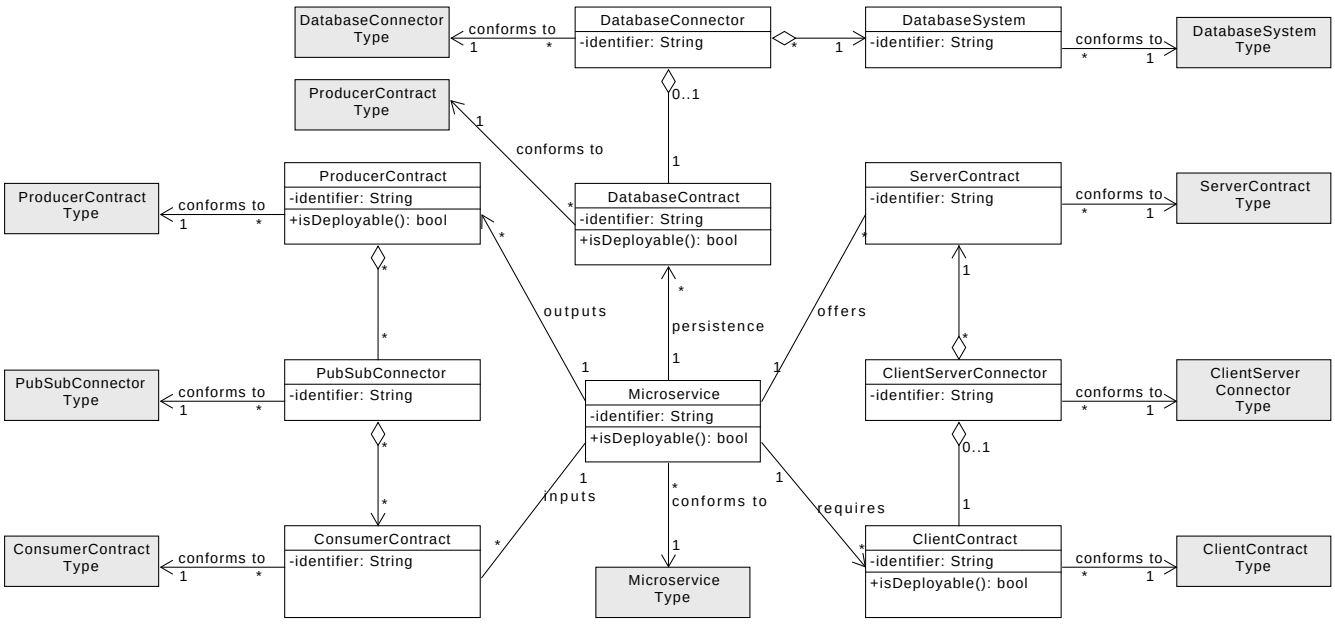

Fig. 3. Model of deployed instances of a configuration

publish-subscribe connector type may have multiple instances. A configuration is composed of a set of microservices, connectors, and database systems. Microservice instances are created from the microservice type, i.e. the type acts as a template such that, for each contract type of the microservice type, a contract instance is created.

In the model of types, the contract types and database types are created first; then, microservice types and connectors are created. In the model of instances, database systems and connectors are created first; then microservices with their contracts are instantiated; and finally, microservice contracts are linked to database systems or connectors.

Whenever there are changes occurring in the current configuration, architects can check whether instances are deployable. A microservice is deployable if and only if (1) its client contracts are deployable, i.e. the contract is attached to a connector when its type specifies that it must have a connector attached, (2) its producer contracts are deployable, and (3) its database contracts are deployable. Everytime a change is committed in the instance model, the deployable property is checked. Therefore, a model of instances represents the topology of the managed system with deployable microservices, connectors and database systems.

IV. Evolution Graph

Configuration types and configurations are organised in a graph as depicted in Figure 4. This graph shows the evolution trajectory of a microservice architecture from the two points of view of types and instances. When version changes are adopted in the code base, architects take a sequence of change actions in the type model, such as adding a new microservice type or a new connector type to the current configuration type. This creates the next configuration type to commit. A configuration type can be committed only if it is instantiable. Once the commit is made, a new node representing a valid configuration type is created in the evolution graph. In the figure, the graph has several branches, for example when a configuration type such as `CT3` is abandoned.

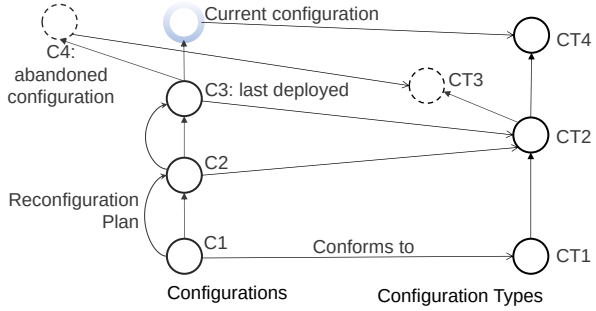

Fig. 4. Evolution graph of configuration snapshots and configuration types

Regarding the configurations from the instance model, each node corresponds to a configuration and has a unique link to a configuration type, i.e. to the configuration type it conforms to. In addition, every configuration node, except the current configuration in preparation, represents a valid configuration, i.e. the configuration is deployable and each commit creates a new node in the evolution graph. Based on the evolution graph, we use an AI planner to generate a reconfiguration plan to transit from a departure configuration node (source architecture) to an arrival configuration node (target architecture). The generated plan can then be transmitted to an executor for

automatic deployment.

Considering using SemVer, we put in action the evolution graph composed of configuration type nodes and configuration nodes into three scenarios: incompatible *major* changes, compatible *minor* changes and impactless *patch* changes. We illustrate the simplest scenario of patch changes as an example. Patch changes are for example due to correcting code bugs or improving implementation, and have no impact on other elements. With the use case of Figure 1, let us suppose that a new version, e.g. `1.0.1`, of the `authentication_service` microservice comes out. The old version `1.0.0` will continue to be supported and may still be instantiated. Architects first create and commit a new configuration type node. Then, they create and commit a new configuration node by choosing which instances are replaced for executing the new version, and which instances keep executing the previous version. As shown in Figure 4, in case of problem when executing the reconfiguration plan, it is possible to return to the previous configuration, which, in the scenario, is based on the previous configuration type.

V. Related Work

Some recent works have discussed multi-version microservice management with contributions complementary to ours.

In [21], the authors propose to build a microservice evolution model by observing the managed system under execution. The model combines together architectural, infrastructure and instance information. It includes versioning of only some of the model elements, namely, application, service, and operation. Similarly, the model in [22] is derived in a bottom-up approach from the concepts of a specific ecosystem of microservice technologies. The two works focus on synchronous interactions between microservices. By contrast, our proposal separates the type model from the instance model and exposes the conformity between the two models. Hence, all the elements of the type model are versioned. In addition, we include database systems and asynchronous communication.

In [7], the authors track microservice dependencies and versions by analysing code (JAVA annotations) and runtime entities (JAVA reflection mechanism), i.e. dependency errors are detected at runtime. The dependency graph is computed through a series of chain searches and version management is based on chain manipulations. Communications can be synchronous or asynchronous. In our work, the dependency graph is included in the model so that errors are detected at design time at configuration commit, and version management is based on model element manipulation and is logged.

In [23], the authors propose a tool to record microservice dependencies at every version update by taking an OS-like package management approach. They create a version timeline per microservice that includes version dependencies to record revision histories, e.g. major or minor updates and dependency requirements about compatible versions of other microservices. In our work, we present microservice relationships in a more detailed manner and we trace evolution histories at the granularity of a configuration, which includes a set of microservices, not just at a single microservice.

VI. Conclusion

In this paper, we discussed how our proposed runtime models and evolution graph can help engineers manage microservice version management, abstract architectural evolution and perform reconfiguration deployment. Our models separate types from instances, and consider both synchronous and asynchronous communication modes (either channel-based or topic-based publish/subscribe systems). In addition, the evolution graph tracks the evolution trajectory of the microservice architecture. In our future work, we plan to refine the models by adding other asynchronous communication modes (such as stream communication), and position technologies such as Kafka that provides more than one communication mode, or such as service mesh platforms that bring into play software defined networking entities to microservice architectures.

References

[1] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly, 2015.

[2] J. Lewis and M. Fowler, "Microservices," https://martinfowler.com/articles/microservices.html, Mar. 2014.

[3] J. Kephart and D. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[4] P. Jamshidi *et al.*, "Microservices: The journey so far and challenges ahead," *IEEE Software*, vol. 35, no. 3, pp. 24–35, 2018.

[5] T. Preston-Werner, "Semantic versioning 2.0.0," *http://semver.org*, 2013.

[6] X. He *et al.*, "Optimal evolution planning and execution for multi-version coexisting microservice systems," in *18th ICSOC*, Dec. 2020.

[7] S. Ma, I. Liu, C. Chen, J. Lin, and N. Hsueh, "Version-Based Microservice Analysis, Monitoring, and Visualization," in *APSEC*, Dec. 2019.

[8] A. Decan and T. Mens, "What do package dependencies tell us about semantic versioning?" *IEEE TOSE*, 2019.

[9] M. Huebscher and J. McCann, "A survey of autonomic computing – degrees, models, and applications," *ACM CSUR*, vol. 40, no. 3, 2008.

[10] Méhus, J.-E. and Batista, T. and Buisson, J., "ACME vs PDDL: Support for Dynamic Reconfiguration of Software Architectures," in *French Conf. on Software Architectures (CAL)*, May 2012, pp. 48–57.

[11] J. Barnes, A. Pandey, and D. Garlan, "Automated planning for software architecture evolution," in *Proc. 28th IEEE/ACM ASE*, 2013.

[12] G. Blair, N. Bencomo, and R. France, "Models@run.time," *IEEE Computer*, vol. 42, pp. 22–27, 2009.

[13] D. Weyns, "Software engineering of self-adaptive systems: an organised tour and future challenges," *in Handbook of Software Engineering*, 2017.

[14] J. Kramer and J. Magee, "The evolving philosophers problem: dynamic change management," *IEEE TOSE*, vol. 16, no. 11, Nov. 1990.

[15] N. Mehta, N. Medvidovic, and S. Phadke, "Towards a Taxonomy of Software Connectors," in *22nd ACM ICSE*, Ireland, Jun. 2000.

[16] L. Lim *et al.*, "Enhancing Context Data Distribution for the Internet of Things using QoC-awareness and Attribute-Based Access Control," *Ann. Telecomm.*, Oct. 2015.

[17] A. Beugnard, J.-M. Jezequel, N. Plouzeau, and D. Watkins, "Making components contract aware," *IEEE Computer*, vol. 32, no. 7, Jul. 1999.

[18] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec, "The Many Faces of Publish/Subscribe," *ACM Comp. Surveys*, vol. 35, no. 2, 2003.

[19] G. Mühl, L. Fiege, and P. Pietzuch, *Distributed Event-Based Systems*. Springer, 2006.

[20] F. Rademacher, S. Sachweh, and A. Zündorf, "Differences between model-driven development of service-oriented and microservice architecture," in *IEEE ICSA Workshops*, 2017, pp. 38–45.

[21] A. Sampaio *et al.*, "Supporting Microservice Evolution," in *33rd IEEE Conf. Software Maintenance and Evolution*, Sep. 2017.

[22] J. Sorgalla *et al.*, "AjiL: enabling model-driven microservice development," in *Proc. 12th ECSA Companion*, 2018, pp. 1–4.

[23] S. Rajagopalan and H. Jamjoom, "App–Bisect: Autonomous Healing for Microservice-Based Apps," in *Proc. 7th USENIX HotCloud*, Jul. 2015.