Supporting Microservice Evolution

Adalberto R. Sampaio Jr.*, Harshavardhan Kadiyala‡, Bo Hu‡,
John Steinbacher†, Anthony Erwin†, Nelson Rosa*, Ivan Beschastnikh‡, Julia Rubin‡

* Federal University of Pernambuco, Brazil ‡ University of British Columbia, Canada † IBM, Canada

***Abstract*—Microservices have become a popular pattern for deploying scale-out application logic and are used at companies like Netflix, IBM, and Google. An advantage of using microservices is their loose coupling, which leads to agile and rapid evolution, and continuous re-deployment. However, developers are tasked with managing this evolution and largely do so manually by continuously collecting and evaluating low-level service behaviors. This is tedious, error-prone, and slow. We argue for an approach based on service evolution modeling in which we combine static and dynamic information to generate an accurate representation of the evolving microservice-based system. We discuss how our approach can help engineers manage service upgrades, architectural evolution, and changing deployment trade-offs.**

I. Introduction

Cloud platforms offer pay-as-you-go resource elasticity and virtually unbounded resources. However, to take advantage of these features, developers must judiciously distribute business logic on the platforms. Microservices [1] are a popular pattern for distributing functionality. A Microservice-Based Application (μApp) is a distributed system that consists of small, loosely coupled, mono-functional services (microservices) that communicate using REST-like interfaces over a network. Microservices are typically developed and deployed independently, resulting in polyglot μApps that rapidly evolve and are continuously re-deployed.

Understanding a single microservice may be straightforward, but μApps often contain dozens of inter-dependent microservices that continuously change. Monitoring and logging stacks for microservices, such as the Elk stack[1], are essential to understanding the microservices in a μApp and are broadly adopted. Unfortunately, logs produced by such stacks contain low-level information for a *single* deployment. Reconciling the view of the deployed version of the system with the historical view of changes being introduced requires interpretation by the developer.

For example, a log may record a failing REST invocation against a particular URL, but it is up to the developer to determine if this invocation was introduced in a recent change and requires fixing or if it indicates an undesirable dependency that should rather be eliminated. Furthermore, non-trivial tasks require piecing together logged information from multiple sources, such as multiple system logs, container infrastructure data, real-time communication messages, and more; collecting and analyzing such information in the context of an evolving system relies on non-trivial knowledge and effort.

[1]https://logz.io/learn/complete-guide-elk-stack/

In collaboration with our industrial partner, IBM, we identified several evolution-related maintenance tasks that are challenging for microservice developers. Supporting these and similar tasks is the focus of our work.

Next, we overview the tasks and briefly outline the challenges they entail.

- **Checking for upgrade consistency.** Microservices are developed and evolve independently, yet the μApp must remain coherent and functional. Determining compatibility and consistency between microservice versions is a continuous challenge for developers. Today, developers manually identify microservice dependencies and either engage with other developers who own that microservice or evaluate the dependency through code inspection.
- **Identifying architectural improvements.** An evolving μApp will experience software architectural corrosion, such as a decrease in cohesion and increase in coupling between related services. Today detecting such architectural problems and evolving microservice architectures are manual and highly involved processes that require global knowledge of microservice inter-dependencies.
- **Evaluating changing deployment trade-offs.** Microservices offer extensive deployment flexibility. For example, two services can be co-located as two containers on the same machine, as two containers in one VM, or as two VMs on the same machine. A poor deployment choice can increase cost, and hurt performance, scalability, and fault tolerance. Furthermore, these decisions must be re-evaluated as the μApp evolves. Today developers evaluate changing deployment trade-offs through trial and error without a systematic strategy nor much tool support.

In this paper, we propose an approach for combining structural, deployment, and runtime information about evolving microservices in one coherent space, which we refer to as *service evolution model*. By aggregating and analyzing information in the model, we aim to provide actionable insights, assisting μApp developers with maintenance and evolution tasks.

In Section III, we introduce the proposed model. We also describe a preliminary design of a system for populating the model by collecting information from a variety of sources, both static and dynamic. In Section IV, we discuss how the information captured in the model helps developers address

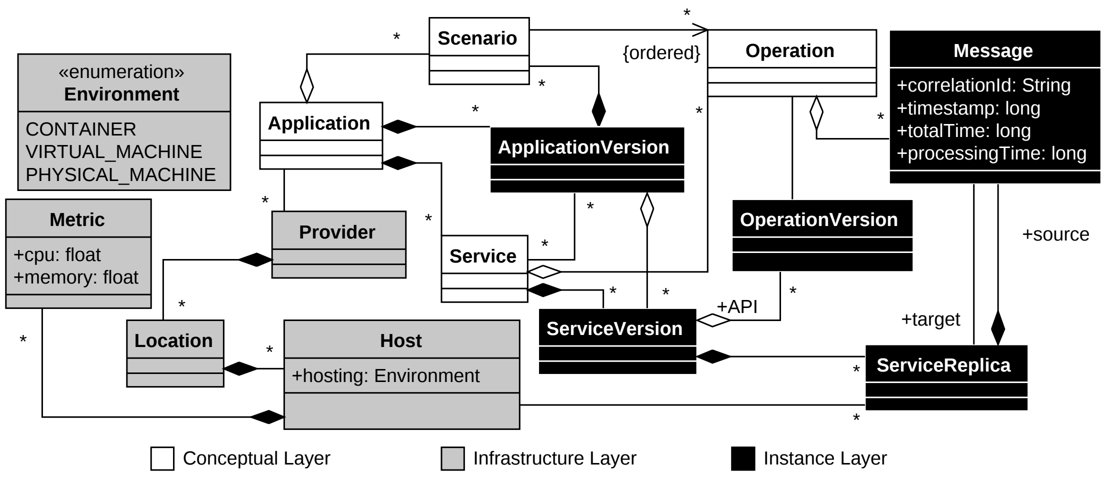

Fig. 1. Service evolution model.

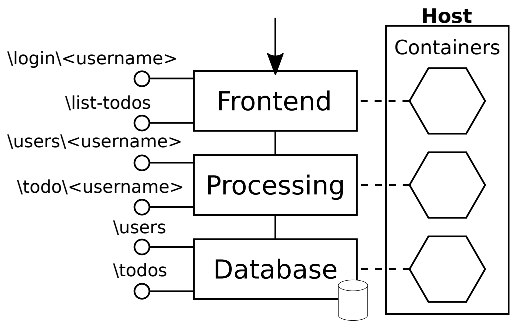

Fig. 2. ToDo application architecture.

the above tasks. Next, we overview the requisite background on microservices.

II. Background on Microservices

High decoupling is a cornerstone of the *microservice pattern* [1], an architectural pattern of *service-oriented computing* [2]. While a consensus has not been reached on what exactly differentiates a microservice from traditional service-oriented architecture (SOA) services [3], most agree that a microservice can be defined as a decoupled and autonomous software, having a specific functionality in a bounded context. Microservices are interdependently managed and upgraded. They communicate using lightweight protocols and are usually deployed inside containers, a lightweight alternative to traditional virtual machines.

The decoupling provided by microservices, together with the agile software delivery and deployment processes [4], [5] decreases the complexity of tasks like upgrades and replication. At the same time, using microservices typically increases the number of interrelated components that make up an application, which creates new consistency issues and poses challenges to evolving microservice-based applications.

To make matters worse, an important feature of microservices is their ability to scale in/out by removing/creating microservice replicas as necessary. This causes the microservices instances to have a short lifetime, inducing further dynamism and complexity.

III. Service Evolution Model

We propose a model for microservices and their evolution in Fig. 1. This model is divided into three layers: the *Architectural* layer (unshaded elements) captures the topology of a μApp. The *Instance* layer (black elements) captures information about service replicas and upgrades, and the flow of μApp messages. This layer links the topology outlined in the *Architectural* layer with deployed microservice instances. The *Infrastructure* layer (gray elements) captures deployment parameters.

Next, we describe each layer (Section III-A) and how we populate the model with concrete information from a μApp deployment (Section III-B).

As our running example we use a simplified version of an open-source *ToDo* μApp application[2] in Fig. 2, which consists of three microservices: *Frontend*, *Processing*, and *Database*, each deployed in its own container. *Frontend* allows new users to log in (via the `\login\<username>` operation) and, for already logged in users, to retrieve the list of their todo items (`\list-todos\<username>`). *Frontend* communicates with the *Processing* microservice to obtain information about a specific user (`\users\<username>`) and to retrieve all todo lists of a specific user from the database (`\todos\<username>`). The database access is managed by the *Database* microservice that provides access to the list of all users (`\users`) and all todo items (`\todos`).

A. Model Description

A μApp is represented by the *Application* element in Fig. 1, which consists of a set of *Services*, each exposing a set of *Operations*. For the example in Fig. 2, the *Frontend* service exposes two operations: `\login\<username>` and `\list-todos\<username>`.

A *Scenario* describes a high-level use case of the application and is realized by an ordered list of operations

[2]https://github.com/h4xr/todo

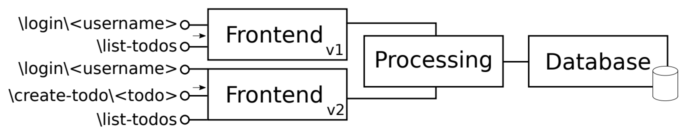

Fig. 3. Two versions of *Frontend*, differing in their supported operations.

executed by services. In the ToDo application, such scenarios are users logging into the application and retrieving their todos. The login scenario is realized by the *Frontend* `\login\<username>` operation, followed by the *Processing* `\users\<username>` operation and *Database* `\users` operation. A scenario specifies the allowed order of operations, helping to detect faulty behaviors: there is no scenario where the *Processing* `\todo\<username>` operation precedes the *Frontend* `\login\<username>` operation.

The *ServiceVersion* and *OperationVersion* elements keep track of changes in services and their interfaces. Any upgrade of a microservice creates a new *ServiceVersion* element. In addition, if the upgrade involves an operation change, a new *OperationVersion* element is created and attached to that new *ServiceVersion* element. For example, adding the `\create-todo\<todo>` operation in the *Frontend* microservice, as shown in Fig. 3, will create new *ServiceVersion* and *Operation* instances.

In blue-green deployments[3], multiple services and multiple versions of the same service can run in parallel, as part of the same application. The *ApplicationVersion* element groups all service versions in a particular configuration. A sequence of *ApplicationVersions* represents the evolution of an application over time.

To model scale-in and out of services, multiple identical instances of a service version are represented by the *ServiceReplica* element. *ServiceReplicas* are *Hosted* by containers or by physical and virtual machines, depending on the *Environment* made available by the cloud *Provider*. Common cloud providers are Amazon AWS, Microsoft Azure, IBM BlueMix, and Google Cloud Platform, each offering several hosting environments.

Hosts can be deployed in multiple geographic *Locations*. Fig. 4 shows a snippet of the deployment model for the *Frontend* service of the ToDo application. In this example, the *Frontend.blue* version is hosted by VM_1 on the East Coast and runs one replica: *Frontend.blue.1*. The *Frontend.green* version is hosted by VM_2 on the West Coast and runs two replicas: *Frontend.green.1* and *Frontend.green.2*. All replicas, on both coasts, correspond to different versions of the *Frontend* service.

To optimize deployment options as the application evolves, we periodically monitor and store *Metrics* related to hosts' CPU load, memory utilization, traffic and latency of requests from a certain area, etc.

A core element of our model is *Message*. Each *Message* represents a uniquely-identified call issued by the source microservice to a particular API (i.e., *Operation*) exposed by the destination microservice. For the example in Fig. 2, the *Processing* microservice exposes the `\users\<username>` operation, which can be called by the `\login\<username>` *Frontend* microservice. Each *Message* carries the timestamp of the request, total time elapsed between issuing the request and obtaining the response, and the time spent in processing the request by each downstream microservice.

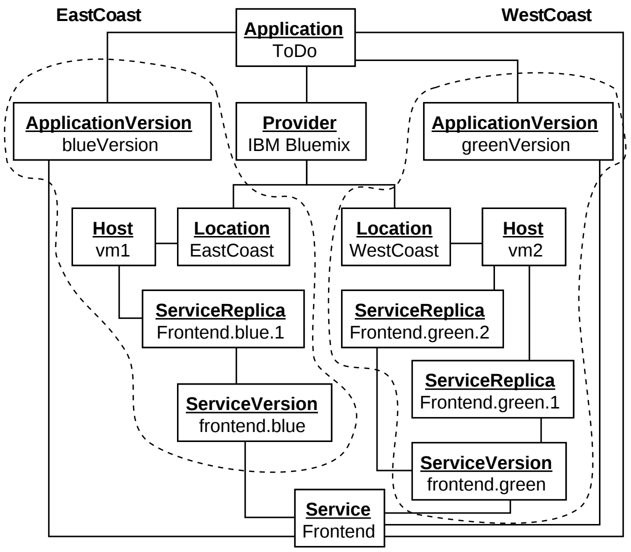

Fig. 4. An example blue-green deployment of the *Frontend* service in Fig. 2.

Messages realizing the same *Scenario* are grouped together via a *correlationId*. For example, when both User A and User B log into the ToDo application, they execute the same login scenario, which involves the same sequence of messages but with different correlation ids: all messages corresponding to the User A login are correlated with each other and are distinct from those of User B.

B. Towards Populating the Model

We generate the model by using information from system logs, container infrastructure data, and messages over protocols like HTTP. More specifically, we extract information about microservices from hosts' meta-data and configuration files, such as deployment files in Kubernetes[4]. To identify operations and their association with services, we rely on a variety of sources: when available, we extract information from API gateways combined with service discovery tools, such as Zuul[5]. We also inspect documentation in tools such as Swagger[6], if that information was published by the developers. We correlate and augment the extracted information by monitoring HTTP messages between services to reveal the used operations.

We generate message elements by using distributed tracing mechanisms, such as Zipkin [6]. We use *correlationIds* in

[3]https://martinfowler.com/bliki/BlueGreenDeployment.html

[4]https://kubernetes.io/
[5]https://github.com/Netflix/zuul
[6]http://swagger.io/

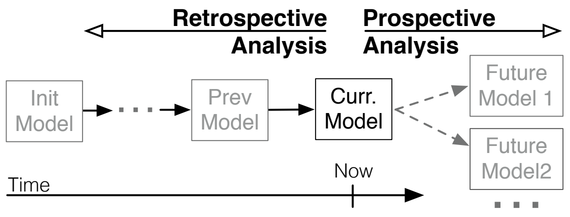

Fig. 5. Retrospective and prospective model analysis.

HTTP requests if developers follow the Correlation Identifier pattern [1]. In case this information is missing, we plan to implement dynamic information flow analysis techniques to correlate input messages with outgoing requests triggered by them.

Scenarios can be identified by grouping requests with the same *correlationId*. In such an implementation, each operation that is the first point of contact for a user will generate a new scenario. To identify scenarios, we plan to analyze test cases associated with an application, with the assumption that all messages generated by a test contribute to one high-level scenario.

By querying cloud provider APIs we plan to extract information about the properties of the provider, such as the data center location, hosts, etc. Interfaces provided by container orchestration systems, such as Kubernetes, can also be used to obtain notifications of new versions and newly created replicas. Performance metrics, like network throughput, CPU, memory, and disk usage can be periodically collected using monitoring mechanism such as cAdvisor[7].

Feasibility. To assess the feasibility of the proposed approach, we implemented an initial prototype of the data collection system on top of Kubernetes, ELK Stack, and an HTTP monitor. One major challenge for our collection and, at a later stage, analysis system is the sheer amount of data that we collect. We intend to utilize graph databases, such as IBM Graph[8], which are designed to store large and complex networks of inter-related data. For host and network metrics, we intend to use data stores built for time series data, such as InfluxDB[9]. Moreover, we intend to periodically compress historical data, keeping only aggregated summaries and statistics.

IV. Evolution use cases revisited

Our generated model captures information about an evolving μApp (Fig. 5). We envision two types of automated model analyses: retrospective (considering current and past models) and prospective (considering current and future models).

Next, we describe how these analyses support the use cases from the introduction: checking upgrade consistency, suggesting architectural improvements, and evaluating deployment trade-offs.

[7]https://github.com/google/cadvisor
[8]https://www.ibm.com/in-en/marketplace/graph
[9]https://www.influxdata.com/

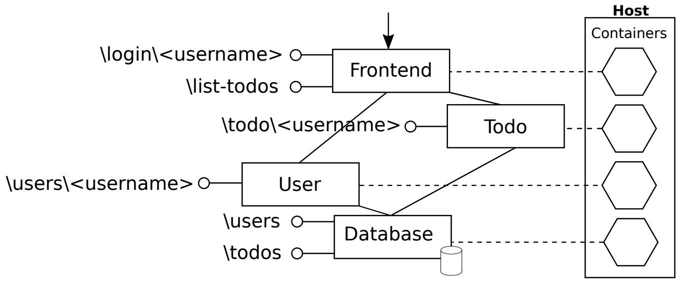

Fig. 6. Refactored ToDo application architecture

Retrospective Analysis. Individual microservices depend on each other to function; a failure in a service might be caused by a change in an entirely different, dependent service. Comparing the sequence of messages in the faulty application scenario to that before the failure occurred, i.e., in the failing and the previous versions of the model, helps detect modified downstream service(s) involved in the scenario. Such services are more likely to be responsible for the fault and should thus be inspected first when *checking for upgrade inconsistencies*.

We can also use retrospective analysis to *recommend architectural improvements*. For example, we can use prior models to identify changes in the μApp topology w.r.t. their communication patterns. Changing coupling and cohesion of services can trigger topology re-organization, e.g., by merging interdependent services.

Splitting "imbalanced" microservices, whose operations exhibit different workloads, can help to scale these operations more accurately. For example, in our ToDo application, once users log in, they create and modify numerous todo items. As such, the login endpoint is underutilized as compared to the endpoint that manages todos. Splitting this microservice into two separate entities, as shown in Fig. 6, makes it possible to scale up the *Todo* microservice while avoiding simultaneous scaling of the *Users* microservice.

The history of metrics stored in our model, correlated with the info on services and their locations, can be used to suggest *deployment improvements*. For example, the *Frontend* and *Todo* microservices in Fig. 6 are tightly-coupled; we can thus suggest that these microservices should be located close to one another. Yet, the *Todo* and *User* microservices do not need such proximity. Likewise, if we observe a sudden decrease in the number of users logging in from a certain geographic location, we can recommend removing the replica at that location, saving money and resources.

In our work, we plan to identify a set of desired architectural and deployment patterns, as in the examples above, and monitor their preservation as the application evolves. That can be achieved by analyzing the collected information on services, operations, messages they exchange, networking and CPU metrics, etc. Whenever the application integrity or quality of service is compromised, our monitoring and analysis system will recommend appropriate improvements, such as replacing a microservice, or moving microservices to different hosts.

Prospective Analysis. Furthermore, we can use our model as a "sandbox" for exploring the space of possible *architectural and deployment refactorings*. We would instantiate several possible refactorings as new snapshots of the model (*Future Models* in Fig. 5) and evaluate their ability to handle the collected real-life μApp scenarios. That is, we will assess potential improvement suggestions by replaying the traces corresponding to the scenarios from the current model in the new model. If the new model withstands a battery of tests, we will issue a recommendation for the change/refactoring to the developers responsible for the relevant microservices.

V. Related work

Evolving architecture has been researched since the notion of software architecture has been articulated. A key approach in this space that combines static analysis, dependency modeling, and evolving architectural concerns is by Sangal et al. [7]. Our approach is similar but targets the microservices domain, which is dynamic and requires runtime analysis.

Work on microservices. There has been increasing interest in applying techniques from the software engineering [8], [9], [10], formal methods [11], and self-adaptive [12], [13], [14] communities to the microservices domain. Our proposal is most similar to *app-bisect* [8] which models the evolution of microservices to help repair bugs in deployment. Our proposal is more general, as it addresses additional evolution-related maintenance tasks, such as deployment and architectural refactorings.

Modeling. Dependency modeling of services is an established topic [15]. Most recently, Düllmann and van Hoorn described a top-down approach to generate a μApp from a model [16]. By contrast, we propose a bottom-up approach that is closer to the work of Leitner et al. [17] and Brown et al. [18]. However, both these approaches only use network interactions between services to generate models and do not model microservice evolution.

Log analysis. Logs are a popular means of monitoring and analyzing software, particularly in the cloud [19]. The state-of-the-art log processing systems are high throughput, real-time, and are capable of reconstructing rich session-level data from logs [6], [20]. Our work builds on these systems.

Supporting microservice evolution. Version consistency has been considered for runtime reconfiguration of distributed systems [21], fault tolerant execution [22], and in other domains. We plan to build on this work and perform upgrade consistency checking at both the model and code levels.

Deployment trade-offs. Previous work considered deployment trade-offs in general distributed systems [23]. Recently Tarvo et al. described a monitoring tool to support canary deployment [10] and Ji and Liu present a deployment framework that accounts for SLAs [24]. We are interested in connecting evolving software engineering concerns with deployment trade-off.

VI. Conclusion

Microservices offer a flexible and scalable means of distributed business logic. However, there are few tools to support developers in evolving microservices and the μApps they comprise. In this paper, we proposed a vision for combining structural, deployment, and runtime information about an μApp to help with evolution-related tasks. Our approach relies on distributed tracing, log analysis, and program analysis techniques and we plan to fully realize and evaluate it in our future work.

Acknowledgments

This works is supported by IBM Canada, CAS program and by CAPES Brazil, grant 88881.132774/2016-01.

References

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

[2] F. Casati, "Service-oriented computing," *SIGWEB Newsl.*, vol. 2007, no. Winter, 2007.

[3] O. Zimmermann, "Microservices Tenets: Agile Approach to Service Development and Deployment," *Computer Science - Research and Development*, vol. 32, no. 3, pp. 301–310, 2016.

[4] M. Hüttermann, *DevOps for Developers*. Apress, 2012.

[5] L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective*. Addison-Wesley, 2015.

[6] "Zipkin," last Accessed: June 2017. [Online]. Available: http://zipkin.io/

[7] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using Dependency Models to Manage Complex Software Architecture," *SIGPLAN Not.*, vol. 40, no. 10, pp. 167–176, 2005.

[8] S. Rajagopalan and H. Jamjoom, "App–Bisect: Autonomous Healing for Microservice-Based Apps," in *HotCloud*, 2015.

[9] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar, "Gremlin: Systematic Resilience Testing of Microservices," in *ICDCS*, 2016, pp. 57–66.

[10] A. Tarvo, P. F. Sweeney, N. Mitchell, V. Rajan, M. Arnold, and I. Baldini, "CanaryAdvisor: a statistical-based tool for canary testing (demo)," in *ISSTA*, 2015.

[11] A. Panda, M. Sagiv, and S. Shenker, "Verification in the Age of Microservices," in *HotOS*, 2017.

[12] S. Hassan and R. Bahsoon, "Microservices and Their Design Trade-Offs: A Self-Adaptive Roadmap," in *SCC*, 2016.

[13] G. Toffetti, S. Brunner, M. Blöchlinger, F. Dudouet, and A. Edmonds, "An Architecture for Self-managing Microservices," in *AIMC*, 2015.

[14] L. Florio, E. D. Nitto, D. Elettronica, I. Bioingegneria, P. Milano, and A. Microservices, "Gru : an Approach to Introduce Decentralized Autonomic Behavior in Microservices Architectures," in *ICAC*, 2016.

[15] C. Ensel, "Automated Generation of Dependency Models for Service Management," in *OVUA*, 1999.

[16] T. F. Düllmann and A. van Hoorn, "Model-driven Generation of Microservice Architectures for Benchmarking Performance and Resilience Engineering Approaches," in *ICPE*, 2017.

[17] P. Leitner, J. Cito, and E. Stöckli, "Modelling and Managing Deployment Costs of Microservice-based Cloud Applications," in *UCC*, 2016.

[18] A. Brown, G. Kar, and A. Keller, "An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in a Distributed Environment," in *IM*, 2001.

[19] A. Oliner, A. Ganapathi, and W. Xu, "Advances and Challenges in Log Analysis," *CACM*, vol. 55, no. 2, pp. 55–61, Feb. 2012.

[20] Z. Chothia, J. Liagouris, D. Dimitrova, and T. Roscoe, "Online Reconstruction of Structural Information from Datacenter Logs," in *Eurosys*, 2017.

[21] X. Ma, L. Baresi, C. Ghezzi, V. Panzica La Manna, and J. Lu, "Version-consistent Dynamic Reconfiguration of Component-based Distributed Systems," in *ESEC/FSE*, 2011.

[22] P. Hosek and C. Cadar, "Safe Software Updates via Multi-version Execution," in *ICSE*, 2013.

[23] R. Guerraoui, M. Pavlovic, and D.-A. Seredinschi, "Trade-offs in Replicated Systems," *IEEE Data Engineering Bulletin*, vol. 39, no. 1, pp. 14–26, 2016.

[24] Z.-l. Ji and Y. Liu, "A dynamic deployment method of micro service oriented to SLA," *IJCS*, vol. 13, no. 6, pp. 8–14, 2016.