DevOps and Quality Management in Serverless Computing: The RADON Approach

Dalla Palma, Stefano; Garriga, Martin; Di Nucci, Dario; Tamburri, Damian A.; van den Heuvel, Willem-Jan

Publication date:
2020

Document Version
Publisher's PDF, also known as Version of record

DevOps and Quality Management in Serverless Computing: The RADON Approach

Stefano Dalla Palma[1], Martin Garriga[1], Dario Di Nucci[1], Damian Andrew Tamburri[2], and Willem-Jan Van Den Heuvel[1]

[1] Jheronimus Academy of Data Science, Tilburg University, Netherlands
{s.dallapalma, m.garriga, d.dinucci, W.J.A.M.vdnHeuvel}@uvt.nl
[2] Jheronimus Academy of Data Science, Technical University of Eindhoven, Netherlands
d.a.tamburri@tue.nl

Abstract. The onset of microservices and serverless computer solutions has forced an ever-increasing demand for tools and techniques to establish and maintain the quality of infrastructure code, the blueprint that drives the operationalization of large-scale software systems. In the EU H2020 project RADON, we propose a machine-learning approach to elaborate and evolve Infrastructure-as-Code as part of a full-fledged industrial-strength DevOps pipeline. This paper illustrates RADON and shows our research roadmap.

Keywords: Infrastructure Code · Serverless Computing · Microservices Computing · Software Quality · DevOps · Machine-Learning for Software Quality · DataOps

1 Introduction

Not so long ago, companies and individuals used to provision and manage their software on their in-house server and computing infrastructure. This behavior implied several costs, such as the costs involved in buying and maintain machines or hiring IT personnel. Therefore, those companies started outsourcing some responsibilities (for example, the management and maintenance of the infrastructure). The cloud has come, which, combined with virtualization and containerization, laid the ground for Infrastructure-as-a-Service, Platform-as-a-Service, and Software-as-a-Service. These technologies allow for more outsourcing and, as a result, a more focus on the business logic while delegating the management of the infrastructure to those having great expertise in the field. Therefore, the lead time has shortened, and the creation of software and its deployment has become relatively easier, cheaper, and quicker. Serverless Function-as-a-Service (FaaS) is a step forward in this evolution.

Function-as-a-service is a serverless way to execute modular pieces of code on the edge. FaaS lets developers write and update a piece of code on the fly, which can then be executed in response to an event, such as a user clicking on an element in a web application. It allows for code scalability and is a cost-efficient

way to implement microservices. Serverless FaaS came with several advantages, among them:

- the business logic is deployed to the host in the cloud as code units in forms of functions, which are fully managed by the provider (e.g., Amazon AWS, Azure Lambda, Google Cloud Functions);
- the lead time, that is, the latency between the initiation and completion of the development process, has shortened;
- less server-side work is needed: the provider is responsible for managing the host machine. Thus, developers can focus on building the business logic.

In general, there are reduced costs and risks. However, serverless FaaS is not a silver bullet, and new challenges arise.

From a survey conducted with serverless adopters, recurrent problems in adopting serverless have emerged [2]. First, serverless is still relatively in its infancy, and hence only a bunch of best practices for its adoption and operations are available. Second, there are few design principles and patterns for composing and triggering serverless functions. Third, few definitions of bad practices exist in the digital-native technical domain, which should be amended.

Furthermore, the rising of serverless solutions, combined with microservices, has forced an ever-increase demand for tools and techniques to establish and maintain them across the entire DevOps lifecycle [1]. With those challenges in mind, the RADON approach for serverless computing comes to play.

2 Objectives

RADON stands for "rational decomposition and orchestration for serverless computing", and is a project funded by the Horizon-2020 European program. It aims to unlock the benefits of serverless computing and Function-as-a-Service (FaaS), and broaden their adoption within the European software industry by developing a model-driven DevOps framework and methodology to create and manage applications based on fine-grained and independently deployable microservices exploiting the serverless paradigm through FaaS and container technologies.

In particular, this work introduces the key tools on the RADON methodology, the user workflow, and their integration and cooperation in the context of DevOps. The methodology strives to tackle complexity, harmonize the abstraction, enforce action-trigger rules, avoid Faas lock-in, and optimize decomposition and reuse through model-based FaaS-enabled development and orchestration.

3 Early Results

We have currently defined the RADON Architecture, including (i) several workflows to organize and display the possible interactions between the tools of the RADON framework and the identified actors and (ii) a tool-chain to define

microservices-based serverless applications with focusing on design-, development-, and run-time.

In particular, RADON envisions a model-based approach to manage and orchestrate modern, distributed, cloud-native application systems that will typically apply a microservice architecture and exploit the FaaS model. The overall RADON framework features rotate around the RADON modeling environment. The other tools are responsible for the correctness and quality of the generated artifacts, such as the Verification, Continuous Testing, and Defect Prediction tools. The framework uses the Topology and Orchestration Specification for Cloud Applications (i.e., OASIS TOSCA) [3] as a baseline to define the RADON models. TOSCA describes the topology generated via the Graphical Modeling tool and the orchestration of cloud applications in a declarative manner. The orchestration process takes place through the Orchestration tool, making it possible to integrate changes and deploy and monitor the application. An overview of the architecture is depicted in Figure 1.

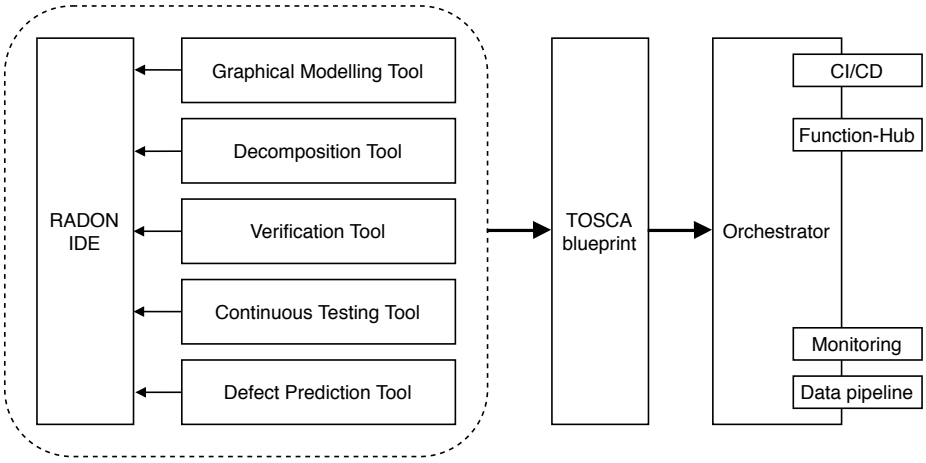

Fig. 1. A broad overview of the RADON framework architecture

3.1 Graphical Modeling Tool

Rather than modeling the environment by manually define TOSCA blueprints, the *graphical modeling tool* (GMT) enables the creation, development, and modeling of TOSCA applications through a web-based software solution. Its main goal is combing TOSCA service templates that represent the applications deployed using the RADON Orchestrator. RADON users can then package their applications as a CSAR (i.e., Cloud Service Archive) before it is deployed into production by using the RADON Orchestrator.

In the first year of the project, the main achievements provided the foundation to graphically maintain RADON applications using the TOSCA Simple Profile standard in version 1.3. Relying on Eclipse Winery as a baseline and by extending it with the respective YAML-based modeling features, allowed a comprehensive modeling tool to graphically (i) create and adapt reusable modeling entities, such as TOSCA Node Types and Policy Types; (ii) compose RADON application structures in the form of TOSCA Service Templates; (iii) enrich existing RADON applications with test-related and performance-specific attributes using TOSCA Policies; and(iv) export a portable archive containing all information to execute the deployment by using the RADON Orchestrator.

3.2 Decomposition Tool

The topology generated by the GMT, or anyone imported into the tool, can be passed as input to the *decomposition tool* to find the optimal decomposition solution for an application based on the microservices architectural style and the serverless FaaS paradigm, taking into account those constraints.

The tool provides suggestions to map abstract components to concrete technologies and adjust the topology itself. For example, to split a monolithic application into microservices or microservices into serverless functions. The feedback from the tool is sent back to the GMT, containing decomposition suggestions and/or the revised TOSCA model.

In the first year of the project, we introduced several modularized approaches to model the performance of applications based on microservices or serverless functions. This tool enables to extend standard languages for IaC, such as TOSCA, with a modeling formalism that describes the behavior of a RADON application and predicts its performance. Such an aspect is essential to obtain optimal deployment schemes.

3.3 Defect Prediction Tool

The same topology can be used by the Defect Prediction tool to analyze the correctness of the delivered infrastructure code.

Indeed, like any other source code artifacts, infrastructure files (such as TOSCA topology definitions or Ansible configuration files) may contain defects that can preclude their correct functioning. The quality of these files should evolve and be maintained through the entire system's life-cycle. The *defect prediction tool* supports the correctness of the infrastructure code developed using the RADON framework and allows DevOps engineers to focus on critical files that may be failure-prone while skipping the others. Thus, allocating resources more efficiently, for example, for testing or code audit.

In the first year of the project, we designed and implemented its architecture to automatically gather meaningful data to improve model performance. More specifically, we developed a set of tools and framework to (i) automatically crawl and mine projects from Github and Gitlab; (ii) extract code and process metrics from Ansible playbooks and TOSCA blueprints; (iii) support DevOps

engineers in training defect-prediction models and identifying snapshots of files containing defects. The tool exposes RESTful APIs that can be used locally or deployed online to interact with the defect prediction tool. The APIs and the MongoDB database will be publicly accessible to retrieve the models trained during our in-vitro experimentations and those added by the community in the future. However, organizations that do not want to expose their data/models can deploy the APIs starting from an empty DB on-premises and grant access to specific users. A command-line client was also developed to use the defect prediction tool in a CI/CD pipeline. It provides functionalities to (i) train a model from scratch using different configurations in terms of data balancing, normalization, feature selection, and classifiers; (ii) download a model from the online APIs; (iii) predict unseen instances based on the model trained with (i) or collected with (ii).

3.4 Verification Tool

The *verification tool* enables a user to verify that a RADON model conforms to a set of constraints (e.g., privacy, security, design pattern violations) before deployment. While modeling the application via the GMT, the Software Designer can set desired properties and constraints (e.g., security/privacy requirements) using a Constraint Definition Language (CDL). The Software Designer can provide examples of the desired behavior through the RADON IDE or manually. Upon the generated models, the software designer or the DevOps engineer can use this tool to perform static checking upon their validity. When a violation is encountered (e.g., circular calls, or privacy violations), the engineer can open the corresponding artifact(s) in the RADON IDE for debugging.

During the first year of the project, the CDL and the command-line version of the verification tool have been developed. It currently supports the verification and the correction of a RADON model and can detect inconsistencies with the constraints expressed in the CDL.

3.5 Continuous Testing Tool

The *testing tool* comprises several modules for microservices/FaaS testing and a data pipeline that will support the continuous testing workflow of RADON. It will provide a set of functionalities to support three main usage scenarios: (i) test case definition, (ii) test execution, and (iii) test maintenance. Such scenarios will help RADON users correctly test their application by creating, executing, inspecting, and removing test cases. In the first year of the project, we specified the requirements of the tool, designed its architecture, envisioned its integration with the other tools of the framework, implemented a prototype, and showcased a proof-of-concept based on two sample applications.

4 Conclusions and Research Roadmap

The RADON project aims to define a decision-making toolkit to optimize micro-services in terms of size, dependencies, and costs by leveraging a reference set of architectural patterns and service templates [1]. To support this goal, the RADON methodology will integrate a workflow that enables decision making on architecture optimization through the *decomposition tool*. To avoid defective infrastructures, it will detect defects through the *defect prediction tool*. To test the application, the framework will check the application invariants against the changes through the *testing tool*. Furthermore, the methodology will define the operations and workflows to use the tools and how they should be integrated and should cooperate. Finally, RADON aims at organizing and accelerating the micro-services evolution in a team-based fashion. Security and privacy policies will be automatically enforced in the run-time environment of the framework to ensure protection for sensitive data and services.

References

1. Guerriero, M., Garriga, M., Tamburri, D.A., Palomba, F.: Adoption, support, and challenges of infrastructure-as-code: Insights from industry. In: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 580–589. IEEE (2019)
2. Lenarduzzi, V., Daly, J., Martini, A., Panichella, S., Tamburri, D.: Towards a technical debt conceptualization for serverless computing. IEEE Software (10 2020)
3. Matt Rutkowski, Chris Lauwers, C.N., Curescu, C.: Tosca simple profile in yaml version 1.3 (2019), `http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/TOSCA-Simple-Profile-YAML-v1.3.html`