Auto-scaling using TOSCA Infrastructure as Code

Matija Cankar[1], Anže Luzar[1], and Damian A. Tamburri[2]

[1] XLAB d.o.o., Pod za Brdom 100, 1000 Ljubljana, Slovenia
https://www.xlab.si/research/
[2] Eindhoven University of Technology - JADS, s'Hertogenbosch, The Netherlands

Abstract. Autoscaling cloud infrastructures still remains a challenging endeavour during orchestration, given the many possible risks, options, and connected costs. In this paper we discuss the options for defining and enacting autoscaling using TOSCA standard templates and its own policy definition specifications. The goal is to define infrastructure blueprints to be self-contained, executable by an orchestrator that can take over autonomously all scaling tasks while maintaining acceptable structural and non-functional quality levels.

Keywords: Cloud Computing · Scaling infrastructures · Autoscaling · TOSCA · Orchestration · Function-as-a-service · FaaS.

1 Introduction

The cloud computing era fostered the emergence of ways to exploit the compute, storage, and network resources and provide new abilities to adapt dynamically the amount of the computing resources to the need of the application that relies on resource provisioning [8]. The aforementioned practice—defined as scaling of the resources [9]—has increasingly became more efficient and responsive to the current application load. On the one hand, two known approaches to scaling are nowadays used, with the first one making possible to *scale-in* or perform vertical scaling, which means that the capacities of the provisioned resources (CPU, RAM, etc.) are scaled up or down. The second is *scale-out* or performing horizontal scaling, which means that the number of units (e.g., virtual machine with specific amount of RAM, CPU, and storage) is enlarged or shrunk.

On the other hand, all cloud providers address the scaling of the leased virtual infrastructure in some way and mostly all do this by their own approach that differs mainly in configuration of this task. For example, the OpenStack [5] uses its own language called Heat, Amazon AWS uses CloudFormation [1] and some cloud management tools, like open-source representative called Slipstream [3] use their own techniques.

Besides the differences of the cloud providers, the approach to the scaling depends on the technology that powers the application. For example, applications based on Docker and Kubernetes allow for horizontal scaling. However,

applications based on Function-as-a-Service (FaaS) [2] will mainly use a combination of vertical and horizontal scaling. Almost all public provides allow manually defined values for vertical scaling, while horizontal scaling is covered by the provider. Only in private open source cloud solutions, as Open-FaaS, both scaling approaches need to be performed by an external software.

To support the autoscalable orchestration of microservice applications intermixing FaaS, Container, and regular virtual-machine components that can be deployed on any provider we develop xOpera, an orchestrator capable of "speaking" with all cloud providers and technologies, addressing autoscaling in a policy-based fashion. On the one hand, interoperability is achieved thorough abstraction, which can be solved with the OASIS standard on "Topology and Orchestration Specification for Cloud Applications" (TOSCA) [4]. On the other hand, the policy-based facilities offered by xOpera only partially address the expected scaling capabilities and the approach shows several limitations.

We contribute to the state of the art and practice with experiences gained through the design and prototypization of xOpera—and its autoscaling features—which we expect to spark fruitful discussions and speedup the standardisation and practice of the auto-scaling definition of cloud applications.

The rest of this paper is structured as follows, first the problem of scaling and current support in TOSCA is presented. In Section 3 presents the concepts of scaling and the Section 4 proposes an approach and evaluation. The paper discusses ideas in Section 5 and is concluded with Section 5.

2 Problem definition

The introduction section described different scaling approaches based on underlying technology. Performing (auto)scaling is complicated task that needs to take into account the state of the application before and after the scaling and perform all required steps to guarantee a safe transition between the states. The software performing this act needs to rely on a set of policies and thresholds that define scaling limits and duties. To perform the scaling on an appropriate moment, the scaling software needs to be subscribed on a messaging queue from monitoring and perform actions with the orchestrator. This tight integration between monitoring service, scaling service and orchestrator service makes the issue even harder and frequently is well solved only inside closed provider environments (Amazon AWS, Microsoft Azure, Google Cloud Platform, etc). This means that user is able to deploy application to one provider and use one approach of scaling, but moving this application to another provider will require manual re-configuration of deployment and scaling. Our aim is to depend on TOSCA standard and propose a way to define scaling at the deploy time, so the orchestrator will be able to create initial deploy on infrastructure and configuration of monitoring in a way that will be easily manageable in case of scaling.

2.1 Definition of application scaling

One most crucial agreement to be defined before forming the scaling approach is to determine *what scaling is* and *what is not*. The modification of the running application is scaling when *we can change the amount of resources without having a significant impact on the running application(s)*. In our case this means that scaling should not result in total un-deploy and redeploy of application but rather moving the functionality from one state to another without shutting down or re-deploying everything.

Based on aforementioned limits we focused on the three most general scaling approaches:

1. simple horizontal scaling, e.g., scaling of containers;
2. simple vertical scaling, e.g., FaaS scaling;
3. complex horizontal scaling, e.g., load balancer example.

In our proposal we consider all other scaling approaches that include more significant changes in the application as (re-)deploying of a new application and are therefore, out of scope.

Next two crucial agreements to be defined on scaling are determining a way to explain the scaling limitations and scaling actions. The TOSCA suggests the use of scaling policies to define scaling limitations and thresholds, as shown on Figure 1. The current version of standard does not provide any details explaining how this data can be used in action therefore it is not obvious what is the job of a scaler, orchestrator or maybe some other tool. When the thresholds are reached something should happen, e.g., a scaler application should take care of it. To overcome this issue we propose a TOSCA which example tries to resolve this issue in section 4.

```
1  tosca_definitions_version: tosca_simple_yaml_1_3
2  my_scaling_policy_1:
3    type: tosca.policy.scaling
4    description: Simple node autoscaling
5    properties:
6      min_instances: <integer>
7      max_instances: <integer>
8      default_instances: <integer>
9      increment: <integer>
```

Fig. 1. TOSCA YAML example from TOSCA v1.3.

3 Scaling concepts

Most common approach to implement scaling is to use three components (see Figure 2), namely orchestrator, monitoring and scaler. In this approach the scaler receives notifications from monitoring, defines next scaling state and instructs the orchestrator how to achieve this change. To describe this approach with a particular monitoring application and a TOSCA orchestrator would imply,

that scaler encapsulates all knowledge about scaling and also about deploying. Scaler would need to a) receive monitoring notifications b) create a new TOSCA-application blueprint based on the defined policies and c) send the blueprint to the orchestrator. The downside of this idea is a division of the process lead between the orchestrator.

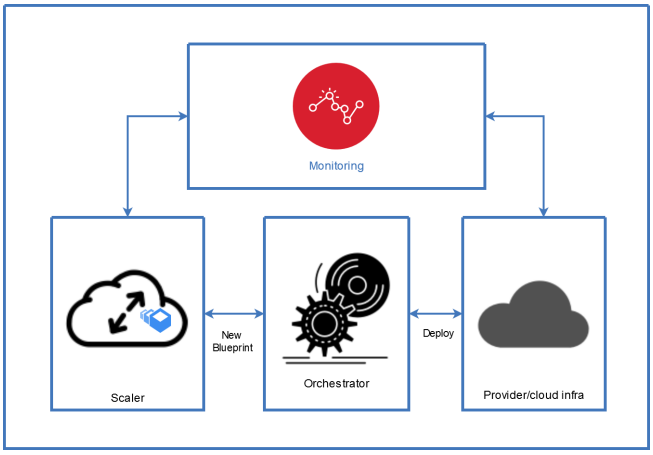

Fig. 2. Scaling using the scaler component.

For example – using the mentioned concept – when orchestrator receives the initial application blueprint in TOSCA CSAR, which we consider that it is an application package including all necessary information to spawn up and set application in runtime, it needs to a) deploy all application components, b) configure monitoring and c) define scaler. But after the initial orchestration of the application is done, the scaler overtakes the leadership and instructs the orchestrator how to proceed. This approach complicates the situations on two levels, first it is a requirement for a scaler, which is TOSCA compliant and can scale applications on various providers. The second issue raises when user sends a small update of the application configuration to the orchestrator which could be overridden by the scaler sending new TOSCA configuration to the orchestrate at the same time. Therefore, if something goes wrong and specifications are different in orchestrator and scaler it is not clear if the targeted application state is in the orchestrator or in the scaler.

4 Proposed approach, scaler inside the orchestrator

Our proposed approach eliminates the scaler and adds its functionality inside the orchestrator, as it is sketched in Figure 3. This means that orchestrator would be the entity in charge of deploying and scaling application at any moment. In the initial deployment orchestrator deploys application, configures monitoring

threshold and defines a TOSCA scaling policy with all scaling definitions. This means that monitoring would send notifications directly to the orchestrator when the thresholds would be reached. In this particular cases, orchestrator would run a `scale` command on the TOSCA blueprint (by executing the linked Ansible playbook) which would perform scaling.

From the Figure 3 it seems that there is no significant change, we anticipate to solve two things. First is the one that the deployment and scaling process deal only with one entity. The second even more important is that the description of the scaling can use the same language as the deploying one and exploits the orchestrator engine and TOSCA actions to perform it.

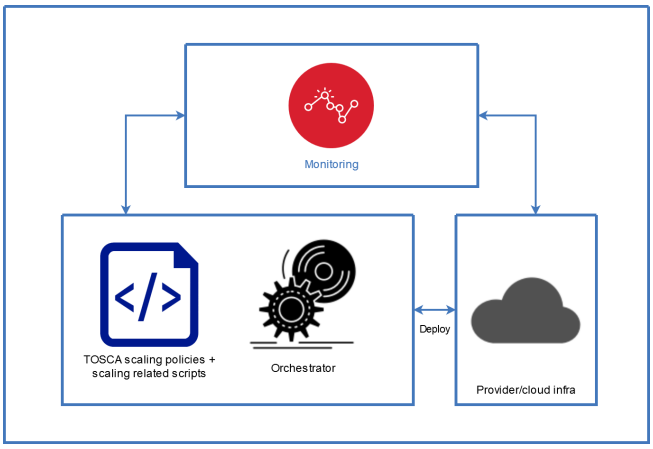

Fig. 3. Proposed architecture without scaler component, TOSCA defines everything and orchestrator is able to accept monitoring notifications.

An example of the proposed TOSCA scaling policy and scaling interface for our approach is shown in Figure 4. Note that the example is simplified to emphasise only the crucial parts of an approach that can be used to fulfill scaling within the TOSCA orchestrator by configuring scaling in TOSCA YAML service template. In the first part of the listing in Figure 4 – lines 1-22 – the OpenStack virtual machine presenting an initial application state is defined. Next block – lines 24-40 – presents the scaling policy, similarly that TOSCA standard suggests and was presented in Figure 1. The property of CPU is defined to be monitored, and the initialisation is globaly limited, not accepting values lower than 80. This threshold should be configured by the orchestrator in a particular monitoring tool as Prometheus. This monitoring configuration part is omitted from our example for brevity. The next part – lines 41-51 – defines the triggers that would initiate scaling procedures. A trigger affects our application, the openstack node, and calls the scaling operation. The important part of this TOSCA definition comes next – lines 53-61 – where we define scale operation called by trigger. This

interface would call a specific Ansible script that orchestrator would process at the deployment phase. This `scale_out.yaml` script would perform scaling on an OpenStack VM e.g. deploy one additional instance to balance the load on the application. The rest of the TOSCA template on Figure 4 – lines 63-79 – defines a topology template that initializes first VM and sets scaling policy properties.

In action the scale-up policy (`radon.policies.scaling.ScaleOut`) would be used when the CPU load would surpass the adjusted value (see `cpu_upper_bound`). When this would happen the policy could use targets keyword to filter out the node it applies to and use a TOSCA scaling trigger definition in order to call the defined TOSCA `scale` operation within the `radon.interfaces.scaling`. This interface operation would then pass on the amount by which to scale (see the `adjustment` parameter) to the Ansible playbook (`scale_out.yaml`) which would perform scaling on an OpenStack VM (for example it could deploy one additional instance so that the load would be balanced).

4.1 Proposed experiment and evaluation plan

The implementation of the proposed concept is in progress and will be finished and tested during RADON project [2]. For the orchestrator we will use xOpera [10] orchestrator with current support of TOSCA v1.3 and Prometheus [6] for monitoring. Currently in progress is finalising the possibility to scale FaaS applications, supporting vertical scaling of the requirements based on configuration update on the providers side, e.g. AWS, Azure or GCP. The next step is to support horizontal scaling with adding or removing container instances. The last step will be to support horizontal scaling of regular virtual machines.

The crucial step here is to define TOSCA types, namely nodes, policies and triggers in a way that will serve as a template to other applications. The outcomes will be tested by the RADON project partners providing template applications and industrial use-cases and the xOpera orchestrator community. The templates for scaling will be published in a publicly available TOSCA template library provided by RADON project and project GitHub repository [7].

Evaluation will be straight-forward with testing the designed TOSCA blueprint of a scalable application with the proposed orchestrator. If this combination will be able to scale application in a same way that scalers can, the verdict will be that the TOSCA standard yaml specification is strong enough to exploit the orchestrator in a way to act as a scaler. That approach simplifies the solution as we do not need special TOSCA scaler to scale TOSCA applications.

5 Discussion and Lessons Learned

Deploying application is a complex job with many tasks. Having the ability to use one IaC language, as TOSCA, is a commodity for a DevOps teams that deploy their applications to heterogeneous environment with multiple cloud providers. The deployment step is very well covered in TOSCA, while the ability to autoscale application is not yet fully defined. During the process of creating the

```yaml
tosca_definitions_version: tosca_simple_yaml_1_3

node_types:
  radon.nodes.OpenStack.VM:
    derived_from: tosca.nodes.Compute
    properties:
      name:
        type: string
      image:
        type: string
      flavor:
        type: string
      network:
        type: string
      key_name:
        type: string
    interfaces:
      Standard:
        type: tosca.interfaces.node.lifecycle.Standard
        operations:
          create:
            implementation: playbooks/create.yaml

policy_types:
  radon.policies.scaling.ScaleOut:
    derived_from: tosca.policies.Scaling
    properties:
      cpu_upper_bound:
        description: The upper bound for the CPU
        type: float
        required: false
        constraints:
          - greater_or_equal: 80.0
      adjustment:
        description: The amount by which to scale
        type: integer
        required: false
        constraints:
          - greater_or_equal: 1
    targets: [ radon.nodes.openstack.VM ]
    triggers:
      radon.triggers.scaling:
        description: A trigger for scaling
        event: trigger
        target_filter:
          node: radon.nodes.openstack.VM
        action:
          - call_operation:
              operation: radon.interfaces.scaling.scale
              inputs:
                adjustment: { get_property: [ SELF, adjustment ] }

interface_types:
  radon.interfaces.scaling:
    derived_from: tosca.interfaces.Root
    operations:
      scale:
        inputs:
          adjustment: { default: { get_property: [ SELF, name ] } }
        description: Operation for scaling.
        implementation: playbooks/scale_out.yaml

topology_template:
  node_templates:
    vm1:
      type: radon.nodes.OpenStack.VM
      properties:
        name: HostVM
        image: centos7
        flavor: m1.xsmall
        network: provider_64_net
        key_name: my_key

  policies:
    test:
      type: radon.policies.scaling.ScaleOut
      properties:
        cpu_upper_bound: 90
        adjustment: 1
```

Fig. 4. TOSCA YAML template example with scaling policy.

framework for developing, deploying and lifecycle management we realised that the latter, which includes scaling, is complex and fragile. Following the approach with *outside scaler* did not promise a stable solution. For example, in case of issues an interruptions during the application life-cycle management, it is not clear which state – scalers or orchestrators – is the desired one? It could be that scaler is in process to submit new application template to the orchestrator, or the orchestrator should update the scalers' configuration.

Exploring the solutions that would not be affected from aforementioned orchestrator-scaler leadership issue we focused on TOSCA definitions and propose the usage of TOSCA, which incorporates scaling definitions to be executed by orchestrator. The TOSCA standard will not require significant updates to use our approach, while some improvements of the TOSCA orchestrators will be necessary. Orchestrator cannot be stopped after the deploy (or particular re-deploy job), but needs to be alive and ready to accept the triggers from the monitoring system. This changes the way how the orchestrator should operate.

6 Conclusions and Future work

To conclude, scaling and autoscaling are and will be desired functionalities within larger microservice applications e.g. AWS Lambda applications and applications using Docker containers. There are several ways of establishing scaling policies successfully but some of them can consume more time or require more tools than others. Instead of commonly used separate scalers this part can be moved to the orchestration process in order to facilitate this task and have universal approach to the application scaling. OASIS TOSCA standard provides promising scaling policies that can be supported in TOSCA orchestrators which can then use their own services to maintain the automatic scaling of application's resources and therefore relieve the end-users and the companies by keeping the applications constantly accessible.

Acknowledgements

This paper has been partially supported by the European Union's Horizon 2020 research and innovation programme under Grant Agreement No. 825040 (RADON).

References

1. Aws cloud formation. https://aws.amazon.com/cloudformation/ (2020), accessed on 2020-6-19
2. Casale, G., Artac, M., van den Heuvel, W., van Hoorn, A., Jakovits, P., Leymann, F., Long, M.C.D., Papanikolaou, V., Presenza, D., Russo, A., Srirama, S.N., Tamburri, D.A., Wurster, M., Zhu, L.: Radon: rational decomposition and orchestration for serverless computing. SICS Software-Intensive Cyber-Physical Systems pp. 1 – 11 (2019)

3. Janiesch, C.: Slipstream: Live dashboarding for sap netweaver bpm ("galaxy"). sap community network blog (2009), http://scn.sap.com/people/christian.janiesch/blog/2009/11/17/slipstream–live-dashboarding-for-sap-netweaver-bpm-galaxy
4. Lipton, P., Palma, D., Rutkowski, M., Tamburri, D.A.: Tosca solves big problems in the cloud and beyond! IEEE Cloud Comput. **5**(2), 37–47 (2018), http://dblp.uni-trier.de/db/journals/cloudcomp/cloudcomp5.htmlLiptonPRT18
5. Openstack heat. https://wiki.openstack.org/wiki/Heat (2020), accessed on 2020-6-19
6. Prometheus. https://prometheus.io/ (2020), accessed on 2020-6-21
7. Radon github repository. https://github.com/radon-h2020 (2020), accessed on 2020-6-29
8. Sahare, D.D.V.R.S.: Cloud computing. International Journal of Trend in Scientific Research and Development **1**(6), 786–789 (Oct 2017), http://www.ijtsrd.com/engineering/electronics-and-communication-engineering/4685/cloud-computing/shubhangi-sahare
9. Wei, Y., Kudenko, D., Liu, S., Pan, L., Wu, L., Meng, X.: A reinforcement learning based auto-scaling approach for saas providers in dynamic cloud environment. Mathematical Problems in Engineering **2019**, 11 (2019), https://doi.org/10.1155/2019/5080647
10. xopera github repository. https://github.com/xlab-si/xopera-opera (2020), accessed on 2020-6-20