Poster: A Declarative Approach for Updating Distributed Microservices

Fabienne Boyer, Nol de Palma
UGA / LIG
fabienne.boyer@imag.fr,noel.depalma@imag.fr

Xinxiu TAO, Xavier Etchevers
Orange Labs
isabelle.tao@orange.com,xavier.etchevers@orange.com

1 BACKGROUND

One of the greatest benefits of microservices is to sensitively ease changing applications by splitting these into independently deployable units [5]. Combined with *Continuous Delivery* (CD) –that aims at delivering quickly and safely every software releases– and *Platform as a Service* (PaaS) automating application management in a on-demand virtualized environment, the microservice paradigm has become essential to implement agile processes.

However, updating microservice applications is still a complex and error-prone task for DevOps teams today, due to the following reasons:

- Usability: most deployment tools adopt a script-based approach [1, 3, 9] for updating microservice applications. As it explicitly describes all required operations, each script is quite complex. Also, despite the use of template, the effort to make it reusable is important.
- Heterogeneity support: some deployment tools are dedicated to a given cloud solution [1, 7]. Nevertheless, for those that support different PaaS providers [3, 9], they provide script templates for each specific PaaS operations. Then, users still have to deal with the complexity resulting from the adaptation of update scripts to the various specific operations exposed by many PaaS solutions.
- SLA requirements: many microservice applications are associated to strict SLA requirements (e.g. availability, time- and cost-performances). Consequently DevOps teams have to define complex scripts to update them in order to minimize downtime and resource cost. Most of these scripts follow well-known strategies such as the *BlueGreen* strategy [2] (minimizing downtime) or the *Canary* strategy [8] (incremental update minimizing resource consumption) [1]. Despite useful, such scripts mix strategies workflow with low-level application management operations. Moreover, the support of a new PaaS provider implies to re-implement all supported strategies.
- Robustness: beyond automation, improving deployment process efficiency also requires it to be able to recover from an arbitrary intermediate state resulting when a failure occurred Most script-base approaches do not address recovery management and each DevOps team implements its solution in its microservice update script. [9] and [3] provide rollback mechanism to recover from a failing update process. However, as rolling back to the initial state is not always the most efficient, roll-forward solutions may be preferred in some cases.

[1]The *BlueGreen* strategy [2] updates a microservice with zero downtime through deploying new microservice instances before uninstalling old ones. In comparison, the *Canary* strategy [8] minimizes the resources needed to process an update but at the expense of a reduced availability, through processing updates *in-place* (the new instances taking the place of the old ones) and incrementally (e.g., site by site).

ICSE'2018, May 2018, Gothenburg, Sweden

ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

2 CONTRIBUTION

We propose a framework easing DevOps teams to update distributed microservice applications. First, we adopt a declarative approach where DevOps teams only need to define the desired target architecture of the application, along with the chosen strategy. To define the application architecture, a DevOps team specifies how microservice instances are spread over the different –potentially heterogeneous– PaaS sites. Second, we propose a simple programming model allowing to customize predefined strategies or to define new ones without having to deal with low-level PaaS specific operations. Third, updates managed with our framework resist to failures thanks to a kill-restart capability.

The runtime of our update framework is mainly organized around two functions: *next* and *push*. The *push* function aims at updating an application A towards a given architecture A' in the most direct way. In order to deal with PaaS-heterogeneity, the *push* function relies on a canonical PaaS interface allowing to introspect, add, modify and remove microservices, following a CRUD (Create, Read, Update, Delete) approach [6]. The *push* function behaves as follows:

(1) it retrieves the current architecture of the application by invoking PaaS canonical interface's introspection operation;
(2) it computes an architectural diff [4] between both architectures A and A', determining the sets of reconfiguration operations (add, remove, modify) to process at each PaaS site to reach the target architecture;
(3) it translates the previous set of canonical operations into sequences of PaaS-specific operations;
(4) it executes the previously computed sequences of operations, in parallel on the different PaaS sites;

The *push* function is strategy-unaware, taking the most direct path for updating microservices. Strategies are taken into account by the *next* function. As illustrated in figure 1, the *next* function goes through the path –made up with intermediate architectures– that allows to reach a given target architecture, according to the chosen strategy. More precisely, a strategy is defined as an ordered list of *architectural transitions* that manage particular elementary updates (e.g., adding, scaling, removing, ... microservices). For instance, we defined the BlueGreen strategy through four transitions:

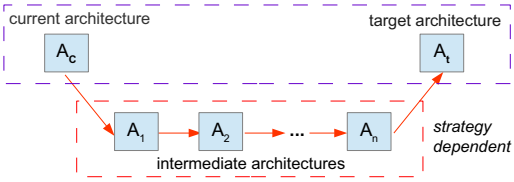

Figure 1: Strategy-driven update

- $Tadd$: deploys microservices defined in the target architecture but undefined in the current architecture.
- $Tupdate$: deploys the green (new) version of the microservices that are modified in the target architecture –associating them to a temporary route (url) for testing purposes–.
- $Tswitch$: switches from temporary route to regular route for green microservices deployed at $Tupdate$.
- $Tremove$: removes microservices that are undefined in the target architecture but defined in the current architecture.

To pass from the current architecture to a desired target one –while conforming to a given strategy–, the proposed framework processes successively all the transitions of the strategy by invoking *next* and *push* sequentially and iteratively. The application is then reconfigured towards each intermediate architecture computed by transitions, ultimately reaching the target architecture. Importantly, the framework provides a preview mode that allows to simulate an update without applying it on the real system, just visualizing the path of intermediate architectures that will be followed.

Figure 2 depicts an example of update process (upgrading M_1 and removing M_3) with the BlueGreen strategy. The transition $Tadd$ has no change to perform. The transition $Tupdate$ delivers the A_1, where M_1 is deployed both in its blue (current) version and in its green (new) version M_1'. Then, the transition $Tswitch$ delivers architecture A_2 where M_1' is now associated to the regular route. Finally, the transition $Tremove$ produces architecture A_3 by removing M_3 and the blue version of M_1. As this resulting architecture matches the target one, the update process is considered to be completed.

Regarding the robustness, the proposed framework provides a kill-restart capability meaning that an update process that previously failed can always be re-started as a fresh update process. Indeed, thanks to our declarative approach,

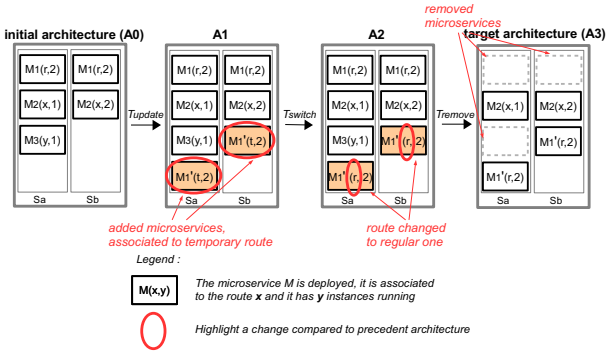

Figure 2: Example update with the BlueGreen strategy

update processes are idempotent because they only invoke the necessary operations to reach the given target architecture. Notice that when re-starting an update process that previously failed, the DevOps team can modify the target architecture (e.g., fixing some microservice confguration, or rollbacking to the initial architecture, or even rollforwarding to a new architecture) and/or the update strategy.

3 CONCLUSION

With the proposed framework, DevOps teams update a microservice application through simply specifying a target architecture and a chosen strategy. They can process an update step by step, possibly changing the strategy or adapting the target architecture at each step. Moreover, they can preview the path of intermediate architecture allowing to reach a given target architecture without impacting the running microservice application. By predicting this path, the proposed framework could, in the future, ease to choose the best fitting strategy for processing a given update according to a set of SLA requirements.

ACKNOWLEDGEMENTS

This work was partially funded by the FEDER project Smart Support Center.

REFERENCES

[1] Amazon Web Services 2018. AWS CodeDeploy official website. (2018). https://aws.amazon.com/codedeploy/
[2] M. Fowler. 2010. Blue-Green deployment. (2010). https://martinfowler.com/bliki/BlueGreenDeployment.html
[3] IBM 2017. IBM UrbanCode official website. (2017). https://developer.ibm.com/urbancode/
[4] J. Kramer and J. Magee. 1990. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE TSE* 16, 11 (1990), 1293–1306.
[5] J. Lewis M. Fowler. 2014. Microservices. (2014). https://martinfowler.com/articles/microservices.html
[6] James Martin. 1983. *Managing the Data Base Environment* (1st ed.). Prentice Hall PTR, Upper Saddle River, NJ, USA.
[7] push2cloud 2016. Push2Cloud. (2016). https://push2.cloud/
[8] D. Sato. 2014. Canary update strategies. (2014). https://martinfowler.com/bliki/CanaryRelease.html
[9] spinnaker 2017. The Spinnaker Continuous Delivery Platform official website. (2017). https://www.spinnaker.io/