Architecture-Based Automated Updates of Distributed Microservices

Fabienne Boyer[1(✉)], Xavier Etchevers[2], Noel de Palma[1], and Xinxiu Tao[2]

[1] UGA/LIG, Grenoble, France
{fabienne.boyer,noel.depalma}@univ-grenoble-alpes.fr
[2] Orange Labs, Paris, France
{xavier.etchevers,xinxiu.tao}@orange.com

Abstract. Microservice architectures are considered really promising to achieve devops in IT organizations, because they split applications into services that can be updated independently from each others. But to protect SLA (Service Level Agreement) properties when updating microservices, devops teams have to deal with complex and error-prone scripts of management operations. In this paper, we leverage an architecture-based approach to provide an easy and safe way to update microservices.

Keywords: Microservices · Dynamic update
Architecture-based reconfigurations

1 Introduction

To facilitate agile development and operations (*devops*), many companies, including established ones such as Netflix [1] and Uber [2], are switching to a microservice architecture for their Cloud applications. With this approach, applications are designed as loosely-coupled services deployed on distributed PaaS (Platform-as-a-Service) sites and running in their own full-stack [3].

The key property that is expected from microservices is the notion of independent replacement and updatability. Especially, microservices exhibit independent lifecycles: they can be deployed and updated independently from each others. The objective is to favor reactivity of small development teams, each team being in charge of developing and evolving its own set of microservices through simple and fast processes.

Such an objective is attractive, but the reality is much more complex because microservices are often associated to SLA properties regarding availability, performances, and resource costs [4,5]. To keep these properties at update time, devops teams follow complex strategies. Typically, the well-known *BlueGreen* strategy [6] intends to update a microservice with zero downtime, but requires deploying and starting all the new microservices before stopping and uninstalling the old ones. In comparison, the *Canary* strategy [7,8] minimizes the resources used at update time, at the expense of a reduced availability: microservices are

C. Pahl et al. (Eds.): ICSOC 2018, LNCS 11236, pp. 21–36, 2018.
https://doi.org/10.1007/978-3-030-03596-9_2

updated *in-place* (new instances taking the place of the old ones), in an incremental manner to slowly transfer the load from the current to the new version.

Using *strategies* to update microservices is considered relevant [9], but so far, the process is managed manually or only automated through using scripts [10]. Scripts provide flexibility but their imperative form limits their ease of use. When devops teams are provided with application-independent scripts, they have to determine what script can be applied to process a given update. Furthermore, they must check that the current state of their application meets the requirements of the chosen script. This is cumbersome and error-prone as most update scripts encompass complex pipelines of PaaS commands. When update scripts are designed specifically for a given application, they can be used in a much easier and safer way, but the price is that devops teams have to compose these scripts, facing the usual coding and debugging challenges.

This paper advocates switching from a script-based to an *architecture-based* approach to automate microservices updates: instead of scripts processing PaaS commands, update strategies are defined as sequences of elementary changes being applied on an *architectural model* of a microservice application. Simply put, this architectural model (also known as model@runtime [11]) reflects how microservices are deployed on PaaS sites and how they are configured. Compared to scripts, the benefits of the proposed approach are the following:

- *ease of use*: to update a microservice application, devops teams simply give as input the desired target architecture, along with the strategy to follow, without having to deal with low-level PaaS commands.
- *preview*: any update can be processed on the architectural model without being applied on the effective system, allowing to preview its result in terms of architectural changes.
- *control*: all stages of an update can be observed on the architectural model. Moreover, at any stage an update can be stopped and resumed with a new target architecture and/or strategy.
- *robustness*: failures occurring at update time are supported.

Leveraging an architecture-based approach raises two main challenges: (i) determining an architectural model encompassing microservices deployed on heterogeneous PaaS sites and (ii) defining a strategy-driven update protocol relying on this architectural model. This paper describes how these challenges were addressed to provide an update framework that can add, remove, migrate, split, or scale microservices as well as upgrade their code or change their configuration across distributed and potentially heterogeneous PaaS sites.

The remaining of this paper is structured as follows. Section 2 summarizes the background. In Sect. 3, we present the architectural model of the proposed update framework. Section 4 describes the strategy-driven update protocol and Sect. 5 focuses on the robustness aspect. An evaluation is given in Sect. 6 and we conclude in Sect. 7.

2 Background

2.1 Microservice Patterns

There is no standard definition for the microservice concept, but common patterns guiding the development of distributed applications on Cloud platforms [12]. We summarize hereafter the patterns that impact the processing of updates.

Microservices are independently deployable modules that run as self-contained units encompassing an operating system along with the necessary runtimes, frameworks, libraries, and code. For improving scalability and availability, each microservice can involve multiple redundant and distributed instances in production.

Microservices communicate through lightweight protocols such as reliable asynchronous bus [8]. They also interact through their provided and consumed services, often exposed as Web services accessed through REST communications. Each instance of a microservice may expose a service through registering a remote API along with a given *route* (*url*) at a registry (usually a per-application registry) such as Consul[1], Apache ZooKeeper[2] or Netflix Eureka[3].

Microservices tolerate the unavailability of the services they access. Two main design patterns are used to this end. Firstly, microservices intend to be stateless through keeping and retrieving any data through an external server, typically a (per-microservice) database. Thereby, any available service instance can be used to execute a given task. Secondly, microservices use smart proxies to access services provided by others microservices [13]. Smart proxies manage the cases where an accessed service is unavailable. Most commonly, depending on the SLA properties of the accessed microservice, a proxy may either (i) select another service instance, (ii) wait for the service to be restored, or (iii) produce a by-default reply, following the circuit-breaker pattern [14,15]. By supporting the unavailability of the services they access, microservices get independent lifecycles—they can be deployed, started, scaled, stopped independently from each other.

2.2 Dynamic Update of Microservice Applications

To update microservices, a first approach is using the Command Line Interface (CLI) provided by PaaS sites [16–18], Although powerful, the CLI is a low-level interface that may be challenging to use directly as most applications are composed of many microservices distributed across PaaS platforms [10].

A second approach relies on using frameworks striving towards continuous delivery features. For instance, Spinnaker [19], AWS CodeDeploy [20] and IBM UrbanCode [21] allow to deploy and update distributed microservices on heterogeneous PaaS platforms. Updating an application goes through a script-based approach where devops teams specify a pipeline of low-level operations to execute. The main limitation is certainly that it is an imperative approach. First,

[1] https://www.consul.io/.
[2] https://zookeeper.apache.org/.
[3] https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance.

devops teams often have to compose the scripts, which faces the usual coding and debugging challenges. Second, they need to check that scripts are compatible with the current state of the updated microservices. Third, they have to make sure that applying such scripts will produce the desired target architecture. Fourth, in case of failures, script-based approaches are usually not idempotent, which requires either to rollback-restart the entire update process, or to analyze the failure to determine how to restart forward, potentially requiring to adapt the scripts.

Another framework, push2cloud [22], allows to update microservices deployed on a single CloudFoundry PaaS site. Recently, push2cloud investigated an approach allowing to express a desired target architecture. However, only mono-site architectures are supported. Moreover, strategies are defined as pipelines of low-level CloudFoundry-specific operations. Finally, failures are fixed and managed manually.

From an academic perspective, [23] automates the deployment of microservices according to a desired target architecture. However, the approach is constrained to using their own dedicated language [24] to program microservices and does not consider update strategies. [10] aims at helping devops teams to manage consistent refactorings, by using a model of a microservice application that covers both architectural and functional aspects. [25] provides an autonomous tool to troubleshoot and repair microservice applications using canary testing and version-aware routing techniques. [26,27] investigate on synthetizing a target architecture for cloud components, considering capacity constraints and conflicts, but they provide no actual mechanism to update running microservices. Other architecture-based approaches for managing reconfigurations [11] are interesting but do not consider any specifics of microservices. In particular, works such as [28–30] have introduced formalisms for automating deployment processes, but they consider components having dependent lifecycles and focus on the management of their dependencies.

3 Architectural Model

With the proposed framework, devops teams update a microservice application by simply giving as input the desired target architecture and the strategy to follow. The strategy may be chosen among pre-defined ones or newly defined.

From a practical point of view, this framework should be launched on a machine having a network connection towards the PaaS sites hosting the microservices to update. Once launched, devops teams can invoke a *pullArchitecture* command to get the current architecture of an application, and a *pushArchitecture* command to update an application towards a given target architecture, following a given strategy, as depicted in Fig. 1.

With both commands, the architecture of an application is expressed through an *architectural model* specifying how microservices are deployed on PaaS sites

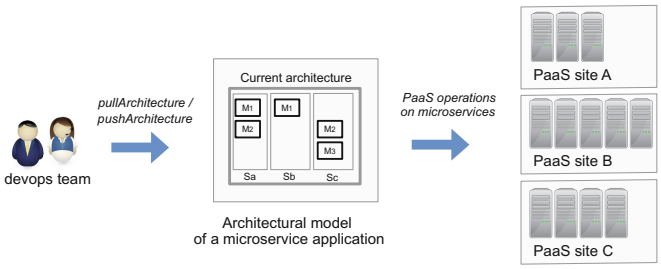

Fig. 1. Operational view

and how they are configured (Listing 1.1[4]). To consider both PaaS-common and PaaS-specific configuration attributes, a microservice is expressed as an extensible set of (attribute, value) pairs[5].

```
Architecture = (String app_id, Set<PaaS−Site> sites);
PaaS−Site = (String site_id, Set<Microservice> services);
Microservice = (String ms_id, Set<String attribute, String value>);
```

Listing 1.1. Architecture model

The architectural model can be introspected and reconfigured through the *PaaSOperations* interface (Listing 1.2). This interface provides four canonical operations allowing to add, get, modify, or remove microservices, following a CRUD (Create, Read, Update, Delete) approach [31]. For any PaaS targeted by the framework, a specific implementation of this interface should be provided, mapping canonical operations towards PaaS-specific operations[6].

```
interface PaaSOperations{
 Set<Microservice> get(String app_id);
 int add(String app_id, Microservice m);
 int remove(String app_id, Microservice m);
 int modify(String ap_id,Microservice m, Set<String attribute, String value>);
 }
```

Listing 1.2. PaaS introspection and reconfiguration interface

Using the *PaaSOperations* interface, our framework can reconfigure a microservice application towards a desired target architecture through the steps given in Listing 1.3. As an example, let's consider a target architecture upgrading a microservice M, deployed on two *CloudFoundry* PaaS sites S_a and S_b, towards a new version $V_{0.2}$ (current version being $V_{0.1}$).

[4] app_{id}, $site_{id}$ and ms_{id} respectively identify a microservice application, a PaaS site, and a microservice.

[5] PaaS-common attributes include *name, code-version, code-path, route, instances-number, lifecycle-state* (STARTED, STOPPED, etc.)).

[6] So far, we mapped this interface for the Cloud Foundry and Heroku platforms.

```
reconfigure(String app_id, Architecture Acurrent, Architecture Atarget) {
 1: compute an architectural diff[27] between Acurrent and Atarget and
    determine the reconfiguration operations (add, remove, modify) to process at each PaaS site
 2: map the reconfiguration operations towards their PaaS−specific implementation
 3: execute the PaaS−specific operations in parallel at each PaaS site
}
```

Listing 1.3. Core reconfigure function

At *step 1*, the *reconfigure* function determines that the following operation should be processed at S_a and S_b:
→ *modify(M,{("code-version", "0.2"),("code-path", "https : //gitX/M/M$_{0.2}$.jar")})*
At *step 2*, the *CloudFoundry* implementation of the *PaaSOperations* interface maps this operation as follow:
→ *cf push M –var version=V$_{0.2}$ -p https : //gitX/M/M$_0$.2.jar*
At *step 3*, this *push* operation is executed in parallel at site S_a and S_b. Notice that such upgrade induces downtime, as the *push* operation stops M, loads its new version of code, and then restarts it, a process taking several minutes in average. The way to avoid downtime is using strategies.

4 Strategy-Driven Updates

A strategy forces an update process to follow a particular path of intermediate architectures, protecting SLA properties throughout the update process, as illustrated in Fig. 2. For instance, still considering the previous upgrade case, the *BlueGreen* strategy would protect the availability property through reaching a first intermediate architecture where M is started in both versions $V_{0.1}$ and $V_{0.2}$, before reaching another architecture where M in version $V_{0.1}$ is stopped.

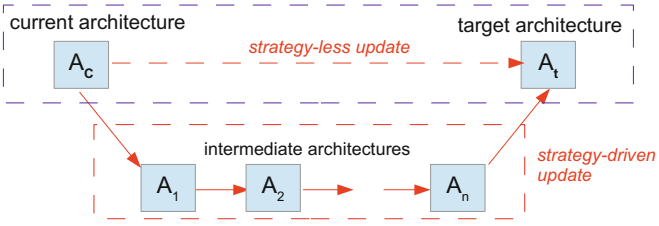

Fig. 2. Strategy-driven update

To account for strategies, we design the *pushArchitecture* command as a *fix-point* (Listing 1.4). At each step, the fix-point requests the strategy to compute the next intermediate architecture along the update path, and then reconfigures the application accordingly.

A strategy is simply defined as a sequence of *transitions*, each transition managing elementary updates. To compute the next intermediate architecture, a strategy goes through its transitions incrementally, until finding one that can

evolve the current architecture closer to the target architecture (see *Strategy* and *Transition* definitions in Listing 1.4).

```
pushArchitecture(String app_id, Architecture Atarget, Strategy strategy) {
  Architecture Acurrent, Anext;
  Acurrent = pullArchitecture(app_id, Atarget.sites);

  while (Acurrent.differ(Atarget)) { /////////////// update fix-point
      // compute next intermediate architecture
      Anext = strategy.nextArchitecture (Acurrent, Atarget);
      if (Anext == null) exit("target unreachable");
      // reconfigure towards Anext
      reconfigure (app_id, Acurrent, Anext);
      Acurrent = Anext;
  }
 }

abstract class Strategy {
     // Sequence of transitions (to define in subclasses)
     List<Transition> transitions;

     // compute the next intermediate architecture to reach
    Architecture nextArchitecture(Architecture Acurrent, Architecture Atarget) {
         // process transitions until finding one moving closer to the target
          for each Transition tr in transitions {
                Architecture Anext = tr.process(Acurrent, Atarget);
                if (Anext != null) return Anext;
           }
          return null;
    }
 }

interface Transition {
    // returns null if the transition does not allow moving closer to the target
     Architecture process(Architecture Acurrent, Architecture Atarget);
 }
```

Listing 1.4. Strategy-driven updates

```
class AddRemoveStrategy extends Strategy {
    // manages additions then removals of microservices,
    List<Transition> = new List(Tadd, Tremove);
}

class Tadd implements Transition {
    Architecture process (Architecture Acurrent, Architecture Atarget) {
    // get microservices added in Atarget compared to Acurrent
    List<Microservice> additions = Atarget.minus(Acurrent);
    if (additions != null) {
     // return an architecture including current miroservices plus the ones to add
        Architecture Anext = Acurrent.clone();
        Anext.add(additions);
        return Anext;
    } else return null;
}}
...
```

Listing 1.5. Example of strategy and transition definitions

For instance, a transition *Tadd* managing the additions of microservices behaves as follow. Comparing the current and target architectures, it determines if new microservices have to be deployed. If yes, it delivers a next architecture containing the current microservices plus the new microservices to deploy. Symet-

rically, a transition $Tremove$ determines if there are microservices to undeploy. If yes, it delivers accordingly an architecture containing the current microservices minus those to undeploy.

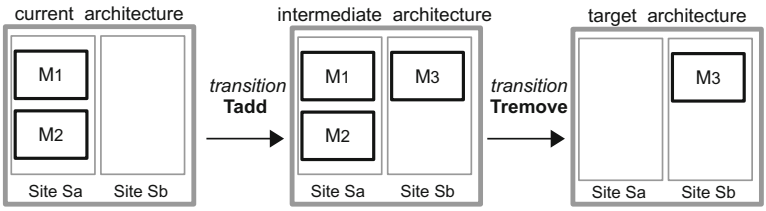

Fig. 3. Using the basic AddRemove strategy

Listing 1.5 illustrates defining a strategy (named $AddRemoveStrategy$) composed of the $Tadd$ and $Tremove$ transitions. Figure 3 depicts the behaviour of this strategy when updating an elementary application composed of two microservices (M_1, M_2) deployed on a site S_a. The target architecture only contains the microservice M_3 on S_b.

- At the first step, the update fix-point processes the first transition ($Tadd$) of the strategy, that delivers the intermediate architecture composed of the current microservices plus M_3 deployed on S_b. The application is then reconfigured towards this intermediate architecture.
- At the second step, the fix-point processes again the transition $Tadd$, that has no more changes to perform. It then processes the next transition ($Tremove$), that removes microservices not appearing in the target architecture (M_1 and M_2 on S_a). The application is then reconfigured towards this architecture and the fix-point terminates because the target has been reached.

Notice that transitions may apply changes over several steps of the fix-point. Let's consider a transition scaling up microservices horizontally as follow. For each microservice to scale, new instances should be deployed and started one by one[7]. Each time it is processed, this transition returns a next architecture in which every microservice to scale has one more instance. When all microservices have reach their target number of instances, it simply returns null.

5 Update Robustness

Two main kinds of failures may interrupt an update process, letting the application in an arbitrary architectural state: first, the framework may faces a hardware failure or a software one, for example a strategy raising an exception when computing the next architecture; second, a microservice may fail when reconfigured

[7] This pattern is required for microservices that do not support having several instances started concurrently.

on a PaaS site, for example it fails to start. With our approach, this is not a problem since any update process may be stopped at any time and later re-started as a fresh update process. This *kill-restart* capability relies on the following main properties.

- *Runnability.* Whatever the current architectural state for a microservice, it can always be introspected and reconfigured by the PaaS site hosting it (even a failed microservice can be restarted by PaaS operations).
- *Idempotence.* Transitions compare the current and target architectures to determine the changes to process. Once a change has been done, a transition just does not do it again. Thus it is always possible to restart an update process that just failed.

Notice that re-starting an update process offers a way to change the target architecture and/or the strategy (Fig. 4), allowing the devops team to fix some microservice configuration or to rollback to a previous architecture for the updated application.

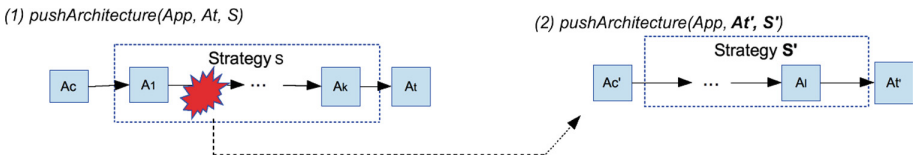

Fig. 4. Management of failures at update time

6 Evaluation

We evaluate our framework on the ease of programming strategies, the ease of updating microservices, and the ability to protect SLAs at update time.

6.1 Ease of Programming Strategies

Let's consider first the *BlueGreen* strategy that updates an application without downtime – through installing, starting, and testing the new version (called the green one) before uninstalling the current version (called the blue one). Once the green environment is ready, incoming requests should be routed to it. We defined such strategy with four transitions.

```
class BlueGreen implements Strategy {
    List<Transition> = new List(Tadd, Tupdate, Tswitch, Tremove);
}
```

Listing 1.6. BlueGreen Strategy

In short, *Tadd* deploys microservices newly defined in the target architecture. *Tupdate* deploys the green version of the microservices that are modified in the target architecture (associating them to a temporary route (i.e., url) for testing purposes). *Tswitch* switches from the temporary route to the regular one for green microservices deployed at the previous step. Finally, *Tremove* removes microservices that are no longer defined in the target architecture. The code of the *Tupdate* and *Tswitch* transitions is shown hereafter (*Tremove* is quite similar to *Tadd* given in Sect. 4). Altogether, these four transitions required only 54 lines of code.

```
class Tupdate implements Transition {
    Architecture process(Acurrent, Atarget) {
        Architecture Anext;
        // get microservices modified in At compared to Ac
        List<Microservice> modified = getModified(Ac, At);
        if (updates != null) {
            Anext = Ac.clone();
            for each Microservice m in modified {
                // deploy green version for the microservice to modify
                Microservice mgreen = m.clone();
                mgreen.route = m.get("temporary-route");
                mgreen.set("role", "green");
                mgreen.set("blue-version", m.get("id"));
                Anext.add(mgreen);
            }
        }
        return Anext;
    } // end of process method
}

class Tswitch implements Transition {
    Architecture process(Acurrent, Atarget) {
        Architecture Anext;
        // get green versions of microservices in current architecture
        List<Microservice> greens = getGreens(Ac);
        if (greens != null) {
            Anext = Ac.clone();
            for each Microservice m in greens {
                    // remove blue version of the microservice
                     Anext.remove(m.get("blue-version"));
                    // assign the regular route to the green version
                    m.set("route", m.get("regular-route"));
                    m.set("role", "blue");
                }
        }
        return Anext;
    } // end of process method
}
```

Listing 1.7. Tupdate and Tswitch transitions

Additionally to the BlueGreen strategy, we programmed a dozen of other classical update strategies[8], some summarized in Table 1. Altogether, they only required programming about fifteen transitions and each strategy was only composed a few transitions (see column named nT). Overall, all transitions were easy to program: (1) they were following similar patterns, essentially comparing the current and target architectures to determine the next architecture, (2) they only required a few lines of code (less than 30).

[8] The code is available at https://github.com/tao-xinxiu/prototype-template-engine.

Table 1. Some strategies programmed

Name	Description	nT
Straight	Reach target directly (no intermediate architecture)	1
CleanRedeploy	Remove all in current, deploy target, one microservice at a time	2
BlueGreen	Reach target, creating green versions then removing old (blue) versions for microservices to update	4
BlueGreenByGrp	As BlueGreen, but processes at most k microservices at a time per site	4
Canary	Reach target, incrementally stopping and restarting instances for microservices to update, site by site	6
CanaryBySite	As Canary, but all instances in parallel on a site	3
CanaryByInst	As Canary, but all sites in parallel	6
Mixed	Reach target, creating one new instance for any microservice to update (for test pupose) then apply Canary strategy for pending instances	5

6.2 Ease of Updating Microservices

We report here on using the proposed framework to deploy and update a microservice application composed of three microservices deployed on two PaaS sites. To perform the initial deployment, we simply use the *pushArchitecture* command with the *Straight* strategy and the desired initial architecture (shown in the left part of the Fig. 5) as target.

Then, to update the application towards the target architecture shown in the right part of the Fig. 5, upgrading M_1 and removing M_3, we decide to use the *BlueGreen* strategy to avoid downtime. The framework allows us to follow step by step the update, through the path of intermediate architectures. When at the architecture 1, we perform some tests, checking that the newly deployed microservice M_1' runs properly on S_a and S_b. Note that external client requests are still routed to M_1, as M_1' is assigned a temporary route.

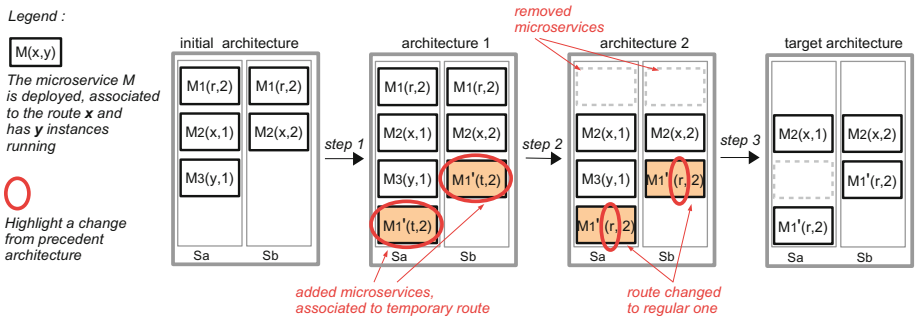

Fig. 5. Elementary application update through the BlueGreen strategy

Once the final target architecture has been reached, we want to continue with a new update, upgrading M_1' towards a new version M_1'' on S_a and S_b. Due to

a faulty configuration, M_1'' fails to run properly on S_b, automatically stopping the update. Here we only have to fix M_1'''s configuration and re-issue the same *pushArchitecture* request to continue the upgrade of M_1' towards M_1''.

However, M_1'''s still fails to run properly on S_b. To fix the problem, we decide to launch a new *pushArchitecture* command with the initial architecture as target, which rollbacks the partial updates we just tried. This results in the automatic un-deployment of M_1'' on S_a and S_b.

Finally, to consider framework failures, we process updates along with a script that randomly kills the framework. This time we consider longer processes, updating one hundred microservices. Each time the framework is killed, whatever the current state of the application, we just have to re-launch it and re-issue the last *pushArchitecture* command to pursue the update towards the desired target.

6.3 Protecting SLA Properties: Real-Life Application Usecase

We used our framework to update a complete clone of a cross-canal order capture application in live production at Orange, focusing on the simultaneous update of two microservices (*(C)atalog* and *(E)ligibility*) that were deployed redundantly over three distributed CloudFoundry (version 2.75.0) PaaS sites (S_0, S_1, S_2). The two microservices are about 10000 lines programmed in Java/Angular. CloudFoundry ran on Cloudwatt [33] on top of OpenStack [34], under VMs with medium flavor (4GB/2VCPU/50GB disk). The update to perform included a code upgrade for the *Catalog* microservice and configuration changes for the *Eligibility* microservice. We experienced with four strategies, comparing their metrics: duration of the whole update process, downtime of the microservices during the update (evaluated with *Apache Jmeter*), and resource consumption (representative of the billing costs for the update). Each experiment was performed 30 times. Results are given in Fig. 6.

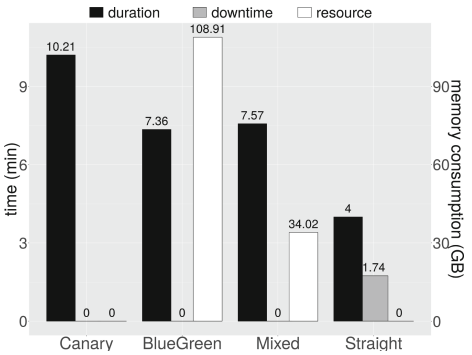

Fig. 6. Update metrics

The *Canary* strategy (whise behavior is shown in Fig. 7) does not involve additional cost in terms of resource usage. It ensures that any deployment of a

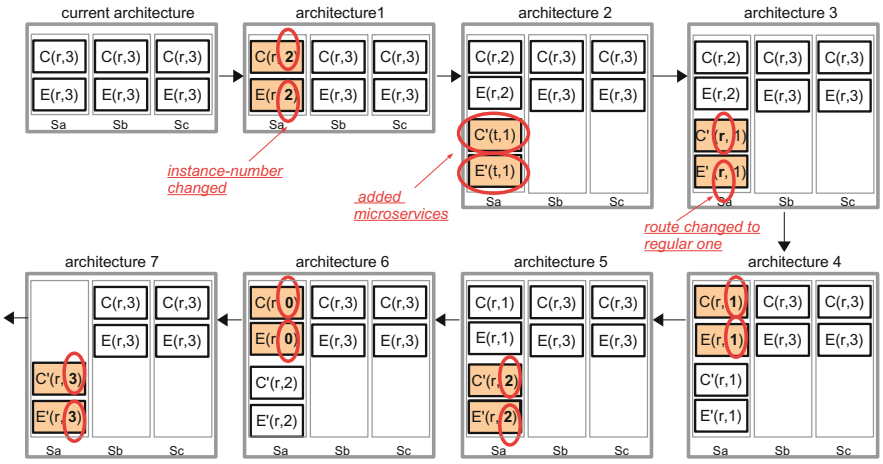

Fig. 7. Real application update through the Canary strategy

new microservice instance is preceded by the removal of a current one. In the version we used, the first new instance being deployed is associated to a temporary route to allow testing before continuing the update. Notice that in case of high request load, some client requests may not be served as the microservices have one less instance running during the update. Overall, this strategy involves 21 intermediate architectures to reach the target, 7 per site, sites being updated sequentially. Its processing took about 10 min.

In comparison, the *BlueGreen* strategy updates the two microservices through creating the three new (green) instances before removing the three current (blue) ones, on all sites in parallel. Accordingly, it ensures zero downtime but consumes nearly the double of memory during the update. The duration is about 7 min, corresponding to the time required to create the new microservices instances (which includes uploading their code), to switch their route, and remove the three blue instances, on one site.

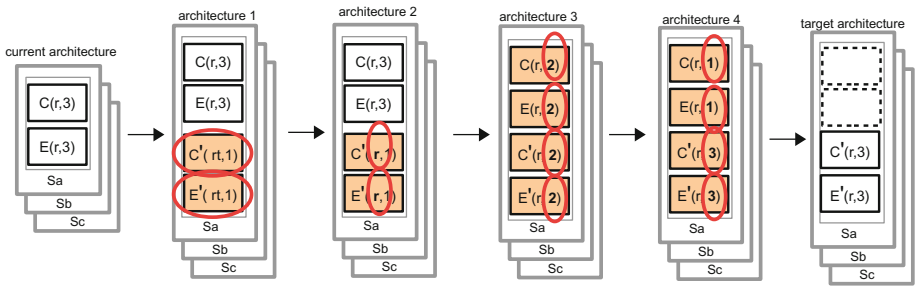

Fig. 8. Real application update through the Mixed strategy

The *Mixed* strategy (Fig. 8) updates microservices instance by instance, creating a new instance before removing an old one, across all sites in parallel. This strategy takes approximately the same duration as *BlueGreen*, with no downtime. It further limits the update cost in terms of resource usage since it uses only one extra instance for each microservice while the *BlueGreen* doubles the number of instances per site.

Finally, the *Straight* strategy delivers the shortest update duration (4 min), as it reaches the target without going through intermediate architectures. The duration corresponds to the time needed to update both microservices on one site, the three sites being updated in parallel. This strategy does not consume any additional resources but induces the largest downtime as the microservices are stopped before their new version is uploaded, recompiled, and restarted.

7 Conclusion

With the proposed framework, devops teams update microservices through specifying target architectures and chosing strategies. They can follow an update step by step, with the opportunity to change the strategy or adapt the target architecture at each step, which is key to handle failures gracefully. Since it permits to pre-compute a path of intermediate architectures, the architecture-based approach leveraged in this paper opens up an interesting perspective: predicting how well a strategy will protect SLA properties during an update.

Acknowledgements. We would like to thank the Orange company for its valuable collaboration in the presented work. This work was partially supported by the FSN HYDDA and the EU FEDER STUDIO VIRTUEL projects.

References

1. Mauro, T.: Adopting Microservices at Netflix: Lessons for Architectural Design. https://goo.gl/DyrtvI
2. Hoff, T.: Lessons Learned From Scaling Uber To 2000 Engineers, 1000 Services, and 8000 Git Repositories. https://goo.gl/1MRvoT
3. Amundsen, M., McLarty, M., Mitra, R., Nadareishvili, I.: Microservice Architecture - Aligning Principles, Practices, and Culture. O'Reilly Media, Sebastopol (2016)
4. Fowler, S.: Production Ready Microservices. O'Reilly, Sebastopol (2016)
5. Carnero, C.: Microservices From Day One: Build Robust and Scalable Software From the Start. Apress, New York City (2016)
6. Fowler, M.: Blue-Green deployment (2010). https://martinfowler.com/bliki/BlueGreenDeployment.html
7. Sato, D.: Canary update strategies (2014). https://martinfowler.com/bliki/CanaryRelease.html
8. Tarvo, A., Sweeney, P.F., Mitchell, N., Rajan, V., Arnold, M., Baldini, I.: Canaryadvisor: a statistical-based tool for canary testing (demo). In: International Symposium on Software Testing and Analysis, ISSTA 2015, pp. 418–422 (2015)

9. Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation, 1st edn. Addison-Wesley Prof, Boston (2010)
10. Sampaio, A.R., et al.: Supporting microservice evolution, In: IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 539–543 (2017)
11. Bencomo, N., France, R., Cheng, B.H.C., Aßmann, U. (eds.): Models@run.time. LNCS, vol. 8378. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08915-7
12. Newman, S.: Building Microservices - Designing Fine-Grained Systems, 1st edn. O'Reilly, Boston (2015). http://www.worldcat.org/oclc/904463848
13. Montesi, F., Weber, J.: Circuit breakers, discovery, and API gateways in microservices, CoRR, vol. abs/1609.05830 (2016)
14. Nygrad, M.T.: Stability patterns. Release It!: Design and Deploy Production-Ready Software, 1st edn. Pragmatic Bookshelf, Raleigh (2007)
15. Fowler, M.: Circuit Breaker (2014). https://martinfowler.com/bliki/CircuitBreaker.html
16. Cloud foundry. http://www.cloudfoundry.com/
17. Heroku. https://www.heroku.com/
18. Openshift. https://www.openshift.com/
19. Spinnaker. https://www.spinnaker.io/
20. AWS CodeDeploy. https://aws.amazon.com/codedeploy/
21. IBM UrbanCode. https://developer.ibm.com/urbancode/
22. Push2Cloud website. https://push2.cloud/
23. Gabbrielli, M., Giallorenzo, S., Guidi, C., Mauro, J., Montesi, F.: Self-reconfiguring microservices. In: Ábrahám, E., Bonsangue, M., Johnsen, E.B. (eds.) Theory and Practice of Formal Methods. LNCS, vol. 9660, pp. 194–210. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-30734-3_14
24. Jolie. Official Web Site. http://www.jolie-lang.org/
25. Rajagopalan, S., Jamjoom, H.: App-bisect: Autonomous healing for microservice-based apps. In: 7th USENIX Conference on Hot Topics in Cloud Computing, HotCloud 2015 (2015)
26. Di Cosmo, R., Lienhardt, M., Mauro, J., Zacchiroli, S., Zavattaro, G., Zwolakowski, J.: Automatic application deployment in the cloud: from practice to theory and back. In: 26th International Conference on Concurrency Theory (CONCUR 2015), vol. 42, Madrid, Spain, pp. 1–16 (2015)
27. Di Cosmo, R., Eiche, A., Mauro, J., Zacchiroli, S., Zavattaro, G., Zwolakowski, J.: Automatic deployment of services in the cloud with Aeolus blender. In: Barros, A., Grigori, D., Narendra, N.C., Dam, H.K. (eds.) ICSOC 2015. LNCS, vol. 9435, pp. 397–411. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48616-0_28
28. Lascu, T.A., Mauro, J., Zavattaro, G.: A planning tool supporting the deployment of cloud applications. In: 25th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2013, pp. 213–220 (2013)
29. Fischer, J., Majumdar, R., Esmaeilsabzali, S.: Engage: a deployment management system. In: 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2012), pp. 263–274. ACM (2012)
30. Etchevers, X., Coupaye, T., Boyer, F., Palma, N.D., Salaün, G.: Automated configuration of legacy applications in the cloud. In: IEEE/ACM Conference on Utility and Cloud Computing (UCC 2011), pp. 170–177 (2011)
31. Martin, J.: Managing the Data Base Environment, 1st edn. Prentice Hall PTR, Upper Saddle River (1983)

32. Kramer, J., Magee, J.: The evolving philosophers problem: dynamic change management. IEEE TSE **16**(11), 1293–1306 (1990)
33. Cloudwatt. https://www.cloudwatt.com/fr/
34. Openstack. https://www.openstack.org/