Reliability of changes in cloud environment at PaaS level

Xinxiu Tao

▶ To cite this version:

Xinxiu Tao. Reliability of changes in cloud environment at PaaS level. Hardware Architecture [cs.AR]. Université Grenoble Alpes, 2019. English. NNT : 2019GREAM001 . tel-02143040

HAL Id: tel-02143040
https://theses.hal.science/tel-02143040

Submitted on 29 May 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTE UNIVERSITE GRENOBLE ALPES

Spécialité : **Informatique**
Arrêté ministériel : 25 mai 2016

Présentée par

Xinxiu TAO

Thèse dirigée par **Fabienne BOYER, MCF, LIG / UGA**
et codirigée par **Noël DE PALMA, Professeur, LIG / UGA**
et **Xavier ETCHEVERS, Ingénieur de Recherche, Orange Labs**

préparée au sein du **Laboratoire LIG et de Orange Labs**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Fiabilisation des mises à jour dans le Cloud au niveau Platform as a Service

Reliability of changes in cloud environment at PaaS level

Thèse soutenue publiquement le **29 janvier 2019**,
devant le jury composé de :

Madame Françoise Baude
Professeur, CNRS, Rapporteur
Monsieur Lionel Seinturier
Professeur, Inria, Rapporteur
Monsieur Daniel Hagimont
Professeur, IRIT/ENSEEIHT, Président
Monsieur Thomas Ledoux
Enseignant-chercheur, IMT Atlantique, Examinateur
Madame Fabienne BOYER
Maître de Conférences, LIG/UGA, Directeur de thèse
Monsieur Noël de Palma
Professeur, LIG/UGA, Co-directeur de thèse
Monsieur Xavier Etchevers
Ingénieur de Recherche, Orange Labs, Co-directeur de thèse

Abstract

Microservice architectures are considered with much promises 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 to perform an update. More precisely, update scripts have to follow particular strategies, such as the the well-known BlueGreen strategy [1] that 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 [2] minimizes the resources used at update time, at the expense of a reduced availability: microservices are 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.

In this thesis, we propose an update framework [3, 4] that leverages an architecture-based approach to provide an easy and safe way to update microservices and program strategies. 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 reflects how microservices are deployed on PaaS sites and how they are configured.

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.

The following of these document 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.

Contents

1 Introduction **1**

2 Problem Position **3**
- 2.1 Microservices 5
- 2.2 Cloud Computing 7
  - 2.2.1 Resources Virtualization 7
  - 2.2.2 Cloud Services Models 8
  - 2.2.3 PaaS/CaaS layers and microservices 9
- 2.3 Continuous Delivery 11
- 2.4 Motivation and Objectives 13
- 2.5 Challenges 15

3 State of the Art **17**
- 3.1 Existing approaches for dynamic software updates 18
  - 3.1.1 DSU 18
  - 3.1.2 Components 18
  - 3.1.3 Actors 19
- 3.2 Existing approaches for dynamic updates of Microservices . . 19
  - 3.2.1 Comparison grid 19
  - 3.2.2 Spinnaker 20
  - 3.2.3 IBM UrbanCode Deploy 23
  - 3.2.4 AWS CodeDeploy 26
  - 3.2.5 Push2cloud 27
  - 3.2.6 Other related approaches for managing Microservices . 29
  - 3.2.7 Summary 30

4 Proposition **34**
- 4.1 Usage Principles 36
- 4.2 Architectural Model 38
  - 4.2.1 Data-Structure 38
  - 4.2.2 Elementary Operations 39
  - 4.2.3 Introspection and Reconfiguration of a Microservice Application 41

4.3 Strategy-driven Updates 45
4.3.1 Update Process Overview 45
4.3.2 Strategy-driven Update Protocol 46
4.4 Strategy Programming 49
4.4.1 Strategy Design 49
4.4.2 Didactic case: the BlueGreen Strategy 51
4.5 Update Robustness 54
4.5.1 Core principles 54
4.5.2 Identification of faults 56
4.5.3 Summary 61

5 Evaluation **64**
5.1 SLA protection 65
5.1.1 Lizard application 65
5.1.2 Account application 68
5.2 Robustness 70
5.2.1 Network faults 72
5.2.2 Update process faults 74
5.2.3 Erroneous strategy 75
5.2.4 Microservice faults 78
5.3 Ease of use 79
5.3.1 Programming Strategies 80
5.3.2 Updating Microservices 81
5.3.3 Comparison with an imperative approach 83

6 Conclusion **86**
6.1 Conclusions 86
6.2 Limitations and Future Works 88

Bibliography **95**

Chapter 1

Introduction

To facilitate agile development and operations (*DevOps*), many companies, including established ones such as Netflix [5] and Uber [6], are switching to a microservice architecture for their Cloud applications. For example, by 2020, Orange aims at migrating about 50% of its production applications –that serve 240 million customers worldwide– to the microservice architecture [7]. With the microservice approach, applications are designed as loosely-coupled services deployed on distributed PaaS (Platform-as-a-Service) sites and running in their own full-stack [8].

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 [9, 10]. To keep these properties at update time, DevOps teams follow complex strategies. Typically, the well-known *BlueGreen* strategy [1] 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 [2, 11] minimizes the resources used at update time, at the expense of a reduced availability: microservices are 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 [12], but so far, the process is managed manually or only automated through using scripts [13]. 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 appli-

cation 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 thesis 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 [14]) reflects how microservices are deployed on PaaS sites and how they are configured. Compared to scripts, the expected 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.

The remaining of this document is organized as follow.

- Chapter 2 introduces the background of the problem, our objectives and the main associated challenges.
- Chapter 3 reports on the related research works about dynamic update, and compares current industrial solutions for managing the updates of microservices application.
- Chapter 4 presents the proposed framework for updating microservices application through an architecture-based approach.
- Chapter 5 presents the evaluation of our framework.
- Chapter 6 concludes this document by summarizing our main propositions and outlining future works.

Chapter 2

Problem Position

Contents

2.1 Microservices 5
2.2 Cloud Computing 7
2.2.1 Resources Virtualization 7
2.2.2 Cloud Services Models 8
2.2.3 PaaS/CaaS layers and microservices 9
2.3 Continuous Delivery 11
2.4 Motivation and Objectives 13
2.5 Challenges 15

Change management is nowadays a well-known term referring to the action of evolving a software system towards a new version, such new version integrating changes in the structure, the code or the configuration of the system. The change management is becoming more and more crucial in the application lifecycle, whatever software development methodology is followed to design, develop and test applications.

The waterfall methodology [15] is a well-known linear software development approach that splits the software development into several well-known phases: analyze requirements, design, code, test and maintain. This methodology promotes spending much time on the early phases to reduce the risk of changes. Anyway, change requirements will appear over time and will need to be managed with minimal impacts on the application.

Compared to the traditional waterfall approach, current popular Agile methodologies [16] promotes embracing changes instead of reducing them. By frequently delivering small changes to end clients, Agile development can better fit to customers' changing requirements, shortening the feedback loop with customers and avoiding the risk inherent in big changes. In this thesis, we address the change management in the specific case of Agile approaches, but the principles of our proposal can be applied in a more general case.

Multiple paradigms help to enforce the Agile methodology:

- **Microservices** application architecture: by decoupling applications into independently-deployable units, microservices intend to reduce the cost of application changes.
- **Cloud** environments: they provide on-demand underlying infrastructure for deploying and executing microservices-based applications.
- **Continuous delivery** discipline: it implies that every change can be automatically released into the production environments.

The rest of this section presents these main paradigms and then explicits the challenges addressed in this thesis.

2.1 Microservices

As previously said, there is no standard definition for microservices, but common guidelines influencing how distributed applications should be designed, developed and managed. Thus, microservices can be considered as a set of architectural patterns for designing applications, aiming at simplifying their deployment and management.

In more precise terms, a microservice-based architecture constructs an application as a set of loosely coupled units called microservice. Each microservice is associated to well-defined business capabilities (e.g., product catalog management, order management). Based on this, each microservice is responsible for managing the data and processing the requests associated to its business capability. More precisely, each microservice may expose its functionalities through *provided services* and may consume services provided by others microservices (i.e. *required services*).

A very important aspect of microservices is their lifecycle independence. Indeed, the microservices of a given application have independent lifecycles in the sense that each microservice can be independently deployed, scaled and updated [17]. To achieve this, each microservice is packaged as a self-contained deployment and execution unit. Such a package encompasses operating system, runtime, frameworks, libraries, and microservice artifact (source code and configuration files).

The lifecycle independence of microservices favors their decentralized governance, a key capability regarding Agile methodologies. Each microservice can indeed be independently developed and operated by a specified small team. Each team is responsible for the technical and technological choices (e.g., programming language, database solution, test and delivery tools) to build its own microservice and do not depend on any other external decision.

To promote this lifecycle independence, three main design patterns are used by microservice architectures:

- The data management in the microservice architecture is decentralized. While monolithic applications prefer a single database for persistent data, microservices prefer letting each service manage its own database instance. Each microservice is in charge of encapsulating, governing, and protecting its own database. This decomposition allows each microservice to choose different data stores based on data shape and read/write access patterns.

- Microservices communicate through lightweight protocols, such as HTTP REST request-response or asynchronous messaging bus. When a microservice instance starts running, it can publish its services as

Web services, and register its address and endpoints at a registry [1], such as Consul [18], Apache ZooKeeper [19] or Netflix Eureka [20]. The microservice instance is unregistered from the registry when it is unavailable [2]. Through the registry, clients of a microservice can know at any time about the available instances of a microservice with the address information allowing to access them.

- Microservices tolerate the unavailability of the services they consume (i.e. accessed services). This capability relies on conforming to dedicated patterns [21] (such as Circuit Breaker, Bulkhead, and Timeout) when designing microservices. Several programming libraries (e.g., Hystrix [22], Finagle [23], Phantom [24]) are available to simplify the development of these patterns. For example, the developer can easily define a fallback function of its accessed service, so that the microservice could get a fallback response when the accessed service is unavailable. In addition, the administrator of the microservice can deploy multiple redundant and distributed instances and use smart proxies to achieve high availability of the microservice. 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.

Overall, through splitting the application into a set of independently deployable services, the microservice architecture minimizes the impacts of changes on a running application. With classical monolithic applications, applying a change to any single module often requires to redeploy the entire application. Conversely, in the microservice-oriented approach, single service changes only imply to redeploy the considered microservice. In other words, the microservice architecture aims at decoupling the application change cycle.

[1] The registry service is usually deployed and consumed by a specific application.

[2] Service registration and unregistration may be either managed by the microservice, or automatically performed by the underlaying PaaS upon starting and stopping the microservice.

2.2 Cloud Computing

Cloud environments enable consumers to run their application in a centrally managed data center. Instead of investing new physical hardware resources to satisfy the requirements of an application, a consumer can only rent the necessary compute, storage, and network resources from the Cloud provider. Thanks to virtualization and autonomous management capabilities provided by cloud environments, consumers can operate the deployment of their application in a self-service manner. In addition, cloud environments help consumers to construct a highly available and scalable application through using redundancy and elasticity techniques at runtime [25].

This section starts by giving an insight of virtualization techniques and technologies. It then introduces the different cloud service models, and finally focuses on the *Platform as a Service* (PaaS) layer specifically, as this layer targets is the one concerned with microservices.

2.2.1 Resources Virtualization

The key technology of cloud computing is virtualization. Through logically separating physical resources into several virtual resources, virtualization facilitates sharing hardware resources. There are currently two main virtualization levels: virtual machine (VM) and container. In addition to dividing system resources, these two techniques can also be used to package an application into a portable self-contained unit. Especially, they are currently both used for packaging and deploying microservices.

In the VM-based approach, hardware IT resources (i.e. compute, memory, storage) are packaged within software units (namely a virtual machine or VM) whose behavior imitates a physical machine environment. The application administrator can package the application running environment into a VM, which contains a full software stack including the operating system (OS), middleware, and the application binaries and configuration elements. Then, the VM can be executed on any physical machine using a hypervisor. Since the VM provides the virtualization at the hardware level, the administrator has the same experience on a VM as on a dedicated physical machine.

The container-based virtualization approach takes place at the operating system level. The OS kernel and possibly some middleware are shared across all the containers running on a common host machine. Accordingly, the application administrator packages only some specific middleware and the application with the container. Therefore, the container includes a quite reduced software stack and does not require a full system boot to be instantiated. Due to their reduced software stack, containers are smaller in terms of size and faster to start up compared to VMs. However, at the opposite, VM-based approaches provide more security because of greater isolation and

fully self-contained. Thus, in order to benefit from the advantages of both approaches, it can be relevant to combine them [26].

2.2.2 Cloud Services Models

According to the NIST's definition [27], cloud environments are divided into three standard service models based on the service abstraction level as shown in Figure 2.1.

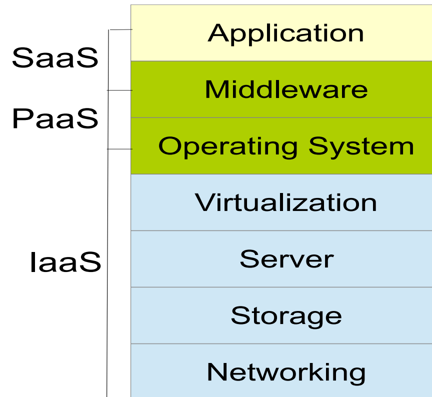


Figure 2.1: Cloud service models. Adapted from [28]

- **Infrastructure as a Service (IaaS)**: at this level, the Cloud provider offers virtualized hardware (e.g., virtualized machines, networks) to the consumer. The control and the responsibility of the application operation are fully given to the consumer. Some examples of IaaS offers are Amazon Elastic Compute Cloud (Amazon EC2) [29], Google Compute Engine (GCE) [30], OpenStack [31].

- **Platform as a Service (PaaS)**: the Cloud provider offers the running environment of the application, the consumers providing only their application code. It is the responsibility of the provider to install the OS and middleware layers for building the application environment. Compared to the IaaS model, the consumer has less control on the infrastructure. Some examples of PaaS offers are Google App Engine [32], AppScale [33], OpenShift [34], Heroku [35], Cloud Foundry [36].

- **Software as a Service (SaaS)**: the Cloud provider completely manages the application. The consumer is the end user of the application. Some examples of SaaS offers are Gmail and Dropbox.

In addition to these three classic service models, **Container as a Service (CaaS)** offerings emerge, such as Google Kubernetes Engine [37], Amazon EC2 Container Service (ECS) [38], Azure Container Service [39].

As a service model positioned between infrastructure-centric IaaS and application-centric PaaS, CaaS is container-centric, meaning that it offers containers as a running environment for an application.

As in PaaS, CaaS also contains orchestration tools (e.g., Kubernetes [40], Apache Mesos [41], and Docker Swarm [42]) for the deployment and cluster management of containers. Moreover, CaaS usually also provides a container image registry and API support for facilitating the management of container runtime. Anyway, unlike PaaS, the CaaS customers stay in charge of packaging their application within containers. An advantage of this is that CaaS doesn't constrain application programming languages or frameworks.

2.2.3 PaaS/CaaS layers and microservices

PaaS and CaaS service layers are intrinsically linked to microservices because they offer an easy way to deploy and run microservice. In the PaaS model, the expected user experience is that PaaS customers are only in charge of their code. In the CaaS layer, the customers have also the responsibility of packaging their code, but the deployment is taken in charge by the CaaS layer. Overall, both layers take the responsibility for installation, configuration, and operation (e.g., routing, monitoring, scaling) of applications composed of microservices.

Typical elementary deployment operations provided by PaaS/CaaS layers are the following:

- *create*: install a microservice (downloading its code, etc.,.)
- *compile*: compile a microservice
- *start*: start a microservice (launching one or more microservice instances)
- *stop*: stop a microservice (stopping one or more microservice instances)
- *scale*: scale a microservice (adding or removing one or more microservice instances)
- *change*: change a microservice's configuration (stopping and restarting it if necessary)
- *remove*: uninstall a microservice

As previously said, a major functionality of PaaS/CaaS layers is to automate the application deployment. Currently, several PaaS solutions (e.g., Heroku, Cloud Foundry, OpenShift) provides a push-to-deploy capability. That is, consumers can deploy their application with a single *push* command.

Especially, PaaS/CaaS customers can easily deploy multiple instances of an application to achieve its high availability. During the execution of applications, PaaS solutions are indeed in charge of maintaining the health of deployed application instances. To this end, PaaS solutions monitor the state of application instances and automatically heal [3] the failed instances. The monitoring mechanism depends on the application types. Web application are classically monitored by sending periodic requests and expecting the responses within a timeout. Applications without connections are monitored by checking the running state of their processes. PaaS customers can configure the healing policy (e.g., restart) for the detected unhealthy instances.

Another common capability of PaaS/CaaS layers is to manage the scaling of applications (while PaaS may manage the scalability in an autonomic way, this is however not yet true for CaaS layers). PaaS customers can horizontally scale their application through configuring the instances number. When the instance number is changed, the PaaS is in charge of deploying or removing the instances. PaaS customers may also vertically scale the disk or memory limit of all the instances. Several PaaS solutions (e.g., OpenShift, GAE) support automatic scaling according to the application workload. The scaling policy is usually based on a set of rules and some application metrics (e.g., CPU utilization, request rate, response latency).

In addition, PaaS layers usually automates the network management of the application. PaaS customers can configure one or multiple URLs for accessing the application. PaaS solutions then take the responsibility of routing the clients' requests from these URLs to a running instance of the corresponding application. PaaS solutions may also provide build-in load balancing across multiple application instances. For example, Cloud Foundry directs the requests in a round-robin [4] manner. To support more complex routing policies, PaaS customers may use an external routing service.

To sum up, although PaaS/CaaS solutions use different implementation technologies, several common capabilities (e.g., deployment, replication, health check, scaling, and network management) are provided in most popular PaaS/CaaS solutions. They provide elementary operations for managing each aspect (usually modeled as a microservice attribute) of a microservice. Because microservices are managed by PaaS/CaaS layers, they naturally benefit from these capabilities.

[3] Depending on the case, PaaS may migrate or restart a application instance

[4] Round-robin means that the router forwards each client request for a given route to each application instance in turn.

2.3 Continuous Delivery

As a software development discipline, Continuous delivery designates the ability to release every application changes into production [5]. In the agile development, continuous delivery is a critical requirement. Without the ability to deliver small and frequent releases to end clients, the benefits of agile development cannot be fully realized. This requirement has lead to the notion of *DevOps teams*, that are composed of a mix of developers and administrators, or at least people that can do both work, for achieving continuous delivery.

The complete delivery process involves building the deployable packages, deploying and testing them in increasingly production-like environments, and finally deploying into production. To improve time efficiency and avoid human errors, DevOps teams need to automate this delivery process. The current best practice is to model the delivery process into a deployment pipeline [43]. Several popular tools are available for setting up deployment pipelines, such as Jenkins [44], Concourse [45], GoCD [46].

The pipeline tool provides the utilities to construct a repeatable process through defining the steps and the connections between these steps. Each step is an elementary job with specified inputs, outputs, and properties. The pipeline tools support multiple types of connection between steps. The steps could be started either automatically as soon as the forward depending steps are completed successfully, or manually after a human approval.

Figure 2.2 illustrates a typical deployment pipeline. This pipeline automatically detects the code change of an application, when the change is detected, it retrieves the code from the Git server, build an image, executes the automatic test, if the test passes, deploys the image to two PaaS sites in parallel, and finally executes a manual test to valid the deployment.

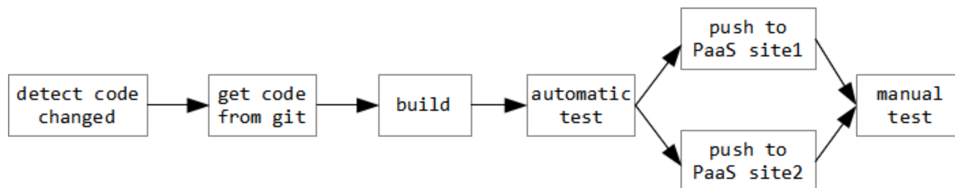


Figure 2.2: An example of deployment pipeline

The pipeline tools give the DevOps teams the visibility and the controllability of the process execution. The pipeline usually provides a graphic interface to visualize the progress of the process execution. In addition, it shows whether the past steps have failed or succeeded. The DevOps teams can start, pause, and stop the pipeline during its execution. The DevOps teams can also define the automated behaviors in case of failures (e.g. stop

[5]Continuous deployment is a more constraining recommendation which requires that every application changes are deployed automatically into production

the pipeline, retry the current failing step, and/or continue with the following steps).

2.4 Motivation and Objectives

To highlight the motivations of this thesis, this section depicts the work that DevOps teams have to achieve to update microservices. Then the objectives of the thesis are described.

Let's consider a common and easy case, updating the code of a single microservice currently running on a PaaS site. As presented in Section 2.2.3, each PaaS/CaaS [6] solution provides its elementary reconfigurations operations for playing a direct in-place update of a microservice. Therefore, based on PaaS operations and the pipeline tools, automating such update requires the following work: write a simple script to call the correct PaaS operations, specify the execution environment of the script in the pipeline tool, and configure the pipeline to be triggered when it detects code changes. The work is quite straightforward until now, relying on the basic reconfiguration operations provided by PaaS solutions.

In the implementation of the update script, the DevOps teams needs to pay attention to the fact that the configuration of a microservice will evolve over time. The update script should not make specific assumption on such configuration. Moreover, if possible, the update script may be used to update various microservices. Thus, it is important to follow a good practice consisting in carefully separating all the microservice-specific configuration from the deployment scripts, so that the script is reusable for various microservices and easier to maintain.

After having implemented a script automating the update of a single microservice on one site, the DevOps teams needs to consider updates encompassing multiple sites. One reason is that the DevOps teams may need to deploy and test a change in multiple production-like environments before deploying it into production. Besides, production environments usually contains multiple sites to achieve high-availability and better performances. An important point is that these sites may use heterogeneous PaaS solutions. Deploying the microservices on heterogeneous PaaS avoids vendor lock-in and allows each microservice to choose independently the most appropriate platform. Therefore, the DevOps teams needs to be able to cope with different PaaS solutions when updating microservices.

Meanwhile, the DevOps teams also needs to consider the updates of multiple microservices on the different sites. Although microservices are supposed to have an independent lifecycle, the reality sensitively differs. In the evolution of applications, it is rarely the case that the contract between microservices always stays unchanged. In addition, the automation of the update of multiple microservices can facilitate the initial deployment and the topology changes.

[6]The rest of the document uses the term *PaaS* to represent *PaaS/CaaS* for simplifying the expression.

Additionally, there are two important and challenging requirements to be taken into account at update time: deployment constraints and failure repairs.

Any deployment or update may be subject to various functional (e.g., a microservice supporting only a single instance running) and non-functional (e.g., minimal downtime, limited resource cost) constraints. As mentioned in Section 2.2.3, the reconfiguration operations provided by PaaS solutions usually perform a direct in-place update of microservice, where the new version takes the place of the old one. This concretely means that a microservice update goes through stopping the microservice instances running the old version, installing the new code version, and re-deploying the new version instances. Therefore, the microservice cannot serve any customers requests during the update.

When the downtime is disallowed due to QoS constraints, the DevOps teams follows complex deployment patterns to update a running microservice These patterns are called *update strategies*, as previously mentioned in this document. For example, to update with minimal downtime, the DevOps teams may choose *BlueGreen* strategy [1]. The *BlueGreen* strategy updates a microservice with zero downtime through deploying the new microservice instances before removing the old ones. However, the *BlueGreen* strategy consumes nearly double resource during the update. To decrease the resource cost, the DevOps teams may choose the *Canary* strategy [2]. The *Canary* strategy requires no additional resources with a reduced microservice performance, through processing in-place updates incrementally (e.g., instance by instance). In these strategies, through manipulating the lifecycle, the accessibility, and the instance number of multiple versions of the microservice, the DevOps teams controls the performance impact, the cost, and the duration of the update. The DevOps teams trades off between these dimensions of non-functional constraints. In practice, although it is not difficult for the DevOps teams to understand the idea of update strategies and choose the proper one for their update, the implementation of strategies is usually cumbersome. The implementation requires correctly controlling and coordinating multiple microservices on multiple sites. Additionally, update processes, and therefore update strategies, have to address the case of failures that occur at update time.

Overall, the work for DevOps teams depicted above demonstrates that it is challenging and error-prone to implement and maintain the automation of updates through a per-application script-based approach. This thesis endeavors to automate update processes in an application-independent manner. In other words, to facilitate implementing continuous delivery of microservices, the objective of the thesis is to propose a framework for efficiently and safely updating microservices deployed on heterogeneous PaaS platforms. The framework aims at supporting various PaaS platforms, various update strategies, and various failure repair operations.

2.5 Challenges

As presented in Section 2.4, the thesis aims at proposing a framework to automate the updates of microservices deployed on distributed PaaS sites. The challenges to address mainly comes from the varieties of the update process regarding four dimensions: change types, PaaS solutions, update strategies, and failure cases.

The first challenge is to support all kinds of changes. Different kinds of changes may be performed in the application updates, mainly: code, configuration, architecture, or topology. The most common changes concern the code and/or the configuration of a microservice. The DevOps teams usually deliver several changes of code and/or configuration daily. Architectural changes mean to update the contract between microservices (such as access interfaces). The delivery of architectural changes is infrequent but more complicated. Such changes usually involve the updates of multiple microservices, requiring coordination between the update operations. Topological changes are relative to the migration of the microservices between different PaaS sites. Their processing involves the coordination between sites.

A second challenge is to cope with heterogeneous PaaS solutions. Each PaaS solution provides its specific microservice model and management operations. Therefore, if the automation of update processes is directly based on PaaS operations, it needs to be reimplemented for each PaaS solution. Using a uniform PaaS interface makes the automation works reusable for different PaaS solutions. However, this option requires the proposed interface to be compatible with various PaaS solutions.

In addition to deploying various changes on heterogeneous PaaS sites, another challenge is to support arbitrary update strategies. The reason is that different updates may impose different non-functional constraints. For example, a security patch is required to be delivered as soon as possible, while an experimental feature may be delivered slowly (e.g., first only delivered to small parts of customers and then propagated to the public). Thus, in addition to providing existing popular strategies, the update framework needs to allow DevOps teams to customize their own strategy.

Finally, the framework needs to properly help DevOps teams to manage (i.e., detecting, fixing) failures during the update. When a failure occur at the update time, the failed update process lets the microservices into an arbitrary state. The difficulty there is to help DevOps teams fixing the failure origin ((e.g., disconnected network, corrupt servers, or erroneous microservice code), and to allow the impacted microservices to recover a functional state rapidly. Although the magic of automatically fixing all the failures is not attainable for the moment, the automation of common repair operations can greatly accelerate the fix and reduce further human errors.

To sum up, the framework needs to automate the updates of multiple

microservices on multiple sites based on the PaaS-provided operations which usually updates a microservice property on one site. In addition, the complex update strategies and failures recovery processes need to be automated. While providing a higher automation level, the framework should still leave the necessary control of the update process to the DevOps teams. Moreover, the framework is expected to be adaptable and extensible for various microservice changes, PaaS solutions, and update strategies.

Chapter 3

State of the Art

Contents

3.1 Existing approaches for dynamic software updates **18**
3.1.1 DSU 18
3.1.2 Components 18
3.1.3 Actors 19
3.2 Existing approaches for dynamic updates of Microservices **19**
3.2.1 Comparison grid 19
3.2.2 Spinnaker 20
3.2.3 IBM UrbanCode Deploy 23
3.2.4 AWS CodeDeploy 26
3.2.5 Push2cloud 27
3.2.6 Other related approaches for managing Microservices 29
3.2.7 Summary 30

Changed customer requirements, fixed software bugs, or new security patches, are all changes that require to adapt the software defining an application, changing its code, its configuration, or the way its components are linked together. Once the new version of the application is defined, applications in production have to be updated accordingly.

To process these updates, an easy way is to shutdown the running old version, then reinstall the new version. However, this imposes some downtime, which does not fit with the requirements of many applications that require 24x7 service availability. To avoid or reduce downtime, dynamic update mechanisms have been considered, aiming at integrating the updates while an application keeps running [47].

This chapter starts by summarizing some main approaches for dynamic update. The objective of this first part is to explain why current approaches for dynamic updates are not directly considered in the microservice world.

In a second part, this chapter presents the approaches that are currently considered for the dynamic updates of microservices.

3.1 Existing approaches for dynamic software updates

According to the targeting application type, the existing research works of dynamic update can be categorized into four groups: DSU, components-oriented, and actors-oriented.

3.1.1 DSU

Dynamic Software Update (DSU) refers to approaches allowing to upgrade a running process (i.e., changing its code to a new version) without any shutdown-restart cycle. Existing DSU techniques [48, 49, 50] rely on writing patches that manipulate core data-structures (e.g., stack, heap, registers) and that rely on low-level capabilities provided by the operating system/runtime to manage code unlinking and relinking.

Overall, such techniques are useful but their limitations prevent using them for microservice updates. Firstly, they are often specific to a given programming language and/or operating system. Secondly, they focus on code upgrade (i.e., evolving the current code towards a new version). Finally, DSU techniques apply for updating a single monolithic component while microservices come with a distributed architecture.

3.1.2 Components

Product component-oriented platforms such as OSGi [51], Spring [52] or Eclipse RCP [53] as well as research prototypes [54, 55, 56, 57, 58] promote applications built from gluing together software components providing and requiring services. Most platforms support dynamic updates through allowing components to be stopped/restarted, created/destructed, and unbound/rebound at execution time.

However, components are more coupled than microservices, potentially sharing data-structures or code packages, and often not supporting the unavailability of the services they depend on. Therefore, in contrast to microservices, components bound together have dependent lifecycles which deeply impacts how updates may be carried out, due to the necessary management of collateral impacts. So far, to the best of our knowledge, component-based update mechanisms have not been adapted to programming patterns of microservices.

3.1.3 Actors

Frameworks such as Erlang [59], Scala [60] or Akka [61] promote *actors* that are small units supporting code updates at runtime through an approach called *design for failure*. This design allows any actor to be killed and restarted at any time. Updating an actor simply goes through killing, updating and restarting it. If other actors are impacted by the unavailability of the updated actor, they can just be killed and restarted as well.

To gain this kill-restart capability, actors developers follow programming patterns that are close to microservices patterns: actors should be stateless, they should only communicate through messages and be memory isolated, they should have an independent lifecycle.

However, actor frameworks impose a particular programming language and runtime. Although a constraint, the benefit is that updating/restarting an actor is fast (in the order of a few milliseconds). In contrast, microservices are heavyweight since they run within their full stack, assembled on-demand by the underlying PaaS platform, which imposes long update/restart times (up to several minutes). Consequently, high-availability concerns require more complex update strategies than a kill-restart approach.

3.2 Existing approaches for dynamic updates of Microservices

Nowadays, several frameworks (mostly industrial ones) intend to help DevOps teams to update their microservice applications, mainly by automating the update process. This section compares the approaches underlaying these frameworks. Section 3.2.1 introduces the grid we will use for the comparison. Then, the following sections presents four main existing automation frameworks: Spinnaker (Section 3.2.2), IBM UrbanCode (Section 3.2.3), AWS CodeDeploy (Section 3.2.4), and push2cloud (Section 3.2.5). Section 3.2.6 summarizes the current research works related to the management (deployment, update) of microservices. Finally, Section 3.2.7 summarizes the related work.

3.2.1 Comparison grid

To compare the usability of the considered frameworks, we analyze the usage effort considering four aspects:

- **automation level**: The automation level reflects the required effort for delivering an update, for the DevOps team. A high automation level requires less effort.
- **flexibility**: Flexibility denotes the capability of the framework to be extended to support specific needs of the DevOps team, especially in

terms of supporting new PaaS platforms, and considering new strategies.

- **reusability**: As presented in Section 2.5, an update process depends on various aspects (change types, PaaS solutions, and update strategies). Reusability denotes the capability of the framework to support updates different update cases.

- **robustness**: Failures inevitably occur during the update. We analyze the effort required to handle failures during the update with each framework, for the DevOps team.

3.2.2 Spinnaker

Spinnaker is an open source continuous delivery platform that help DevOps teams to automate the update of applications across multi-clouds.

Regarding the supported Cloud platforms, Spinnaker supports IaaS providers (AWS EC2, Google Compute Engine, Microsoft Azure, and OpenStack), PaaS providers (Google App Engine), and container orchestrators (Kubernetes, Google Container Engine).

Spinnaker enforces the principle of immutable infrastructure. Once the application is deployed, any change is disallowed on its infrastructure (virtual machines). An update process always deploys a new version of microservices on a new infrastructure. Therefore, Spinnaker does not support in-place updates, where new versions of microservice take the place of older versions.

Spinnaker models an update process as a pipeline. The pipeline consists of a sequence of (reconfiguration) steps. A step, called as a *stage* in Spinnaker, evolve an architectural aspect of a microservice (e.g., an attribute, a number of instances, its lifecycle state, etc.,.). The DevOps team is in charge of constructing the pipeline by composing the steps provided by Spinnaker.

Each step is more precisely either an elementary action or a composed action. For each supported Cloud platform, Spinnaker provides specific elementary actions allowing to deploy, destroy, resize, enable or disable one microservice on one cloud platform [1]. Spinnaker let the DevOps team construct the necessary pipelines when multiple microservices across multiple clouds have to be updated.

Besides elementary actions, an update process may also involve other operations or events, which are called *utility actions*. Spinnaker provides several utility actions (e.g., manual judgment, wait, run Jenkins job) to help linking the microservice management actions when constructing the pipeline.

[1]The action *enable* (resp., *disable*) controls a microservice to start (resp., stop) receiving client requests.

For common action compositions, Spinnaker packages them as built-in update strategies to make them easy to be reused in different pipelines. Regarding the update strategy, Spinnaker provides four update strategies:

- BlueGreen [2]: deploy a new version, if health check passed, scale down and disable old version.
- Highlander: deploy a new version, when it is up and healthy, all old versions are removed automatically.
- RollingPush: gracefully delete old version instances, and replace them by new version instances.
- Canary: deploy a new version, send a small part of traffic to the new version, test the new version, if passed, scale up the new version. Finally, disable and remove the old version.

Automation level

This section reports on the usage effort for delivering a specific update with Spinnaker. Spinnaker requires the DevOps team to construct a pipeline for automating an update process.

Let's consider a DevOps team managing two microservices (M_1 and M_2) on two sites ($site_1$ and $site_2$). The following part of the section reports on how the DevOps team automates the process of updating the microservice (M_1) deployed on two sites ($site_1$ and $site_2$) with the *BlueGreen* strategy.

In practice, the DevOps team first needs to create the model of the microservice M_1 by specifying its metadata, such as name, code location (a git repository), and cloud sites accounts. Then, the DevOps team creates a pipeline as shown in Figure 3.1.

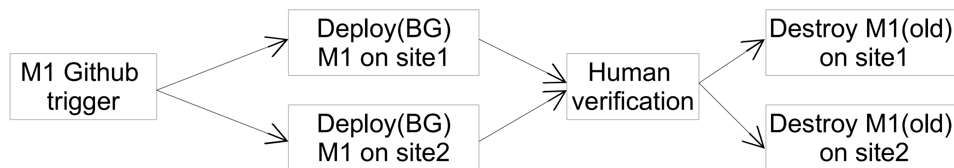


Figure 3.1: Spinnaker example pipeline: update M_1 on $site_1$ and $site_2$ with *BlueGreen* strategy

This pipeline includes a trigger step (*M1 Github trigger*) so that it can be automatically started when M_1's code is updated. Then, the steps for deploying M_1 on $site_1$ and $site_2$ with BlueGreen (BG) strategy are specified to be executed in parallel. After these steps, the DevOps team added a step of human verification. Finally, the DevOps team added the steps for removing the old version.

[2] The BlueGreen strategy is called as Red-Black in Spinnaker

Flexibility

As Spinnaker lets DevOps teams compose the update pipeline, DevOps teams can orchestrate an update process in their own way. However, DevOps team have to compose the pipeline with the steps templates provided by Spinnaker.

In terms of cloud platforms, it is a challenging work to support a new one in Spinnaker. It requires the DevOps team to map the model of a microservices application, implement the elementary actions and update strategies, and integrate these with Spinnaker modules. Currently, Spinnaker does not provide any documentation for supporting new cloud platforms.

Regarding extending the set of strategies, the DevOps team can define a strategy by specifying a sequence of steps, but this sequence should target one microservice on one site. Notice that currently, Spinnaker only supports customized strategies on limited cloud platforms (Amazon Web Service and Google Compute Engine). In addition, Spinnaker does not provide a comprehensive document for customizing a strategy.

Reusability

The previous pipeline (Figure 3.1) updates the code of the microservice M_1. This pipeline also works in case of updating microservice configurations. Thanks to the immutable principle, updating any changes requires the same actions workflow. By simply modifying the configuration file [3] specified in the steps *deploy(BG)* in Figure 3.1 and starting the pipeline, the pipeline can update M_1 to the new version of the configuration. Therefore, Spinnaker pipelines can be reusable for various changes in case of updating the same microservice on the same site.

However, the Spinnaker pipeline is hardly adaptable to various architectural changes and topological changes. The architectural (resp. topological) changes involve the update of multiple microservices (resp. sites). As demonstrated, each step is in charge of updating a specific microservice on a specific site. Therefore, the architectural changes and topological changes require the DevOps team to construct specific pipelines. In addition, it is complex to directly reuse a Spinnaker pipeline for another microservice, as microservice-specifics and site-specifics parameters are integrated into each step of the pipeline.

To sum up, a Spinnaker pipeline is hardly reusable across microservices and sites. This disadvantage may be improved by a feature under development: pipeline templates [63]. However, given that the definitions of elementary actions are specific to each cloud provider, a reusable pipeline across cloud providers will still be difficult to be implemented.

[3] Besides the static file, the DevOps team can also specify the microservice configuration as a dynamic input of the pipeline [62].

Robustness

Spinnaker automates the rollback of a microservice update. It provides the elementary step *rollback* by re-enabling the old version and disabling the failed new version for a given microservice.

Moreover, Spinnaker supports re-triggering a failed step to fix some failures. For the failures which occur during a non-idempotent action (e.g., deploy), it is usually complex to automatically resume the process from an intermediate state. In such cases, an easier solution is to perform a rollback, then re-trigger the update pipeline.

Taken the previous example, the DevOps team can construct a more robust update pipeline as shown in Figure 3.2. When the new version is detected as failed in the step *human verification*, the microservice is automatically rolled back to its old version.

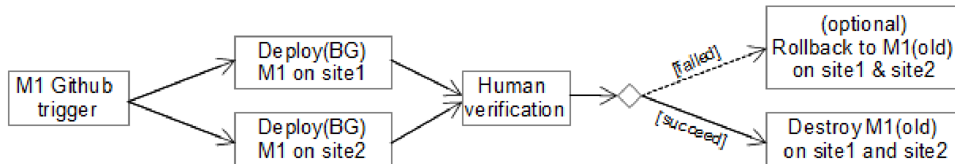


Figure 3.2: Spinnaker example pipeline: update M_1 on $site_1$ and $site_2$ with automatic rollback

3.2.3 IBM UrbanCode Deploy

IBM UrbanCode Deploy (UCD) is a proprietary commercial tool for automating application deployment on multiple clouds [4]. Regarding supported cloud platforms, UCD supports both IaaS providers (IBM SoftLayer, Amazon Web Services, Microsoft Azure, VMware vCenter, and OpenStack-based clouds) and PaaS providers (IBM Bluemix).

UCD provides a graphical flowchart tool to construct pipelines for automating update processes. Similar to Spinnaker, an update pipeline consists of (reconfiguration) steps. The DevOps team is in charge of constructing the pipeline by composing the steps.

In UCD, each step is either provided by the DevOps team (as a custom script) or by third-party plug-ins. UCD provides a great number of plug-ins. For example, *Git* plugin enables the DevOps team to automatically retrieve microservice artifacts from a Git source-code management repository. The *Cloud Foundry* plugin provides the Cloud Foundry command line utility to manage a microservice on a target Cloud Foundry platform.

In terms of update strategies, UCD does not provide pre-defined strategies.

[4]UCD calls a *microservice* as a component, a site as an *environment*

Automation level

We use the example described in Section 3.2.2 (updating M_1 on $site_1$ and $site_2$ with *BlueGreen* strategy) to consider the usage effort for automating a specific update process.

The DevOps team first needs to create the model for the microservice M_1 by specifying its properties (e.g., name, source code repository). Then, the DevOps team can construct the pipeline that updates M_1 with the BlueGreen strategy, as shown in Figure 3.3. In this pipeline, the DevOps team uses the steps provided by the UCD Cloud Foundry plug-in. This plug-in provides Cloud Foundry operations for managing microservice attributes.

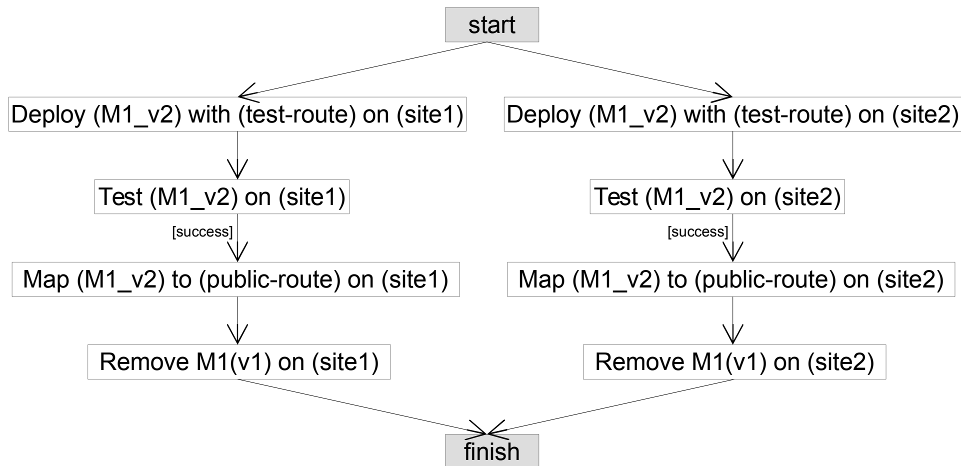


Figure 3.3: UCD example pipeline: update M_1 on $site_1$ and $site_2$ with *BlueGreen* strategy

Flexibility

Similar to Spinnaker (presented in Section 3.2.2), UCD also lets the DevOps team orchestrate the update process. Meanwhile, UCD makes it easier for the DevOps team to define custom steps. Indeed, the DevOps team can easily define a step as a script.

In terms of cloud platforms, UCD allows the DevOps team to support a new platform by defining a dedicated plug-in. UCD provides documents and examples for implementing such a plug-in.

Regarding update strategies, UCD lets the DevOps team define strategies through the concept of process template. A process template is a sequence of steps taking the managed microservices and associated cloud sites as dynamic input.

Reusability

UCD allows a process template to be reused in different pipelines. However, as the steps definition is specific to a cloud solution, a process template is not reusable across cloud solutions.

For example, taken the previous example pipeline (Figure 3.3), the DevOps team can define a process template as shown in Figure 3.4. This process template updates a microservice on a site with the *BlueGreen* strategy. It takes the model of the microservice to update and its associated site as input. Based on this process template, the previous pipeline can be simplified as shown in figure Figure 3.5.

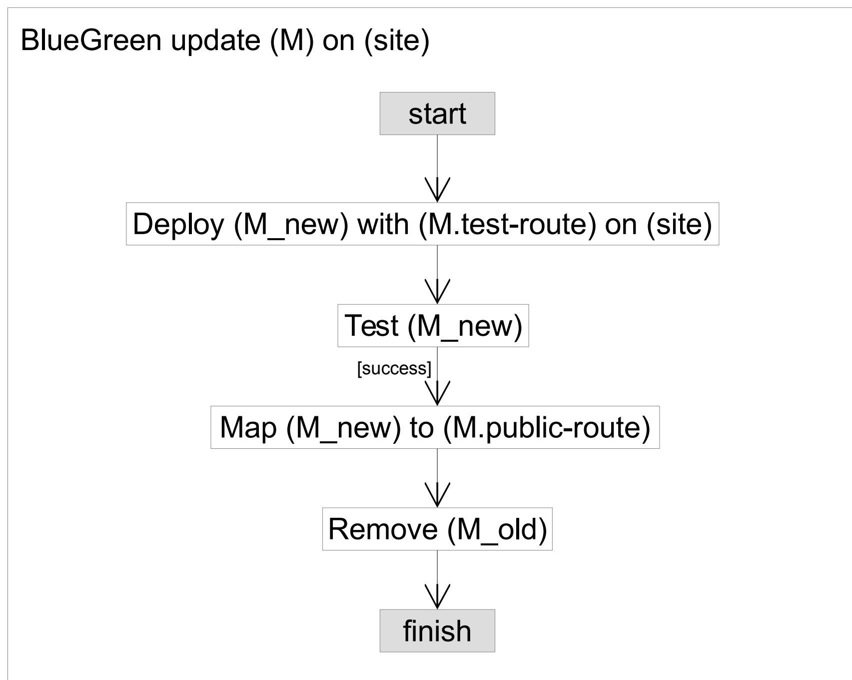


Figure 3.4: UCD example process template: update a microservice on a site with *BlueGreen* strategy

Robustness

UCD helps the DevOps team to handle update failures through retry and rollback mechanisms. The DevOps team can add automated tests in the update process to verify if a step succeeded or failed. In addition, the DevOps team can specify the steps to process in case of failure or success of a previous step.

If a failed update leaves the microservices deployment into an intermediate state, the DevOps team can rollback the failed microservices, then re-run the update process from the beginning. UCD provides two types of rollback

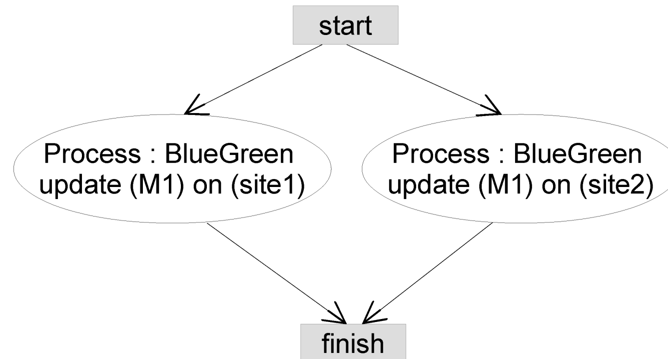


Figure 3.5: UCD example pipeline based on the BlueGreen process template

steps: rollback to a snapshot or rollback a failed update. The process of rolling back to a snapshot uses the snapshot to determine the microservice versions to roll back to, and removes the microservice versions which are not defined in the snapshot. The process of rolling back a failed update redeploys the previous version of the microservice before the execution of the update process.

3.2.4 AWS CodeDeploy

AWS CodeDeploy is a deployment service that automates microservice deployment on Amazon EC2 instances. This is a solution specific to the *Amazon Web Service* cloud.

AWS CodeDeploy supports three types of update strategies: in-place update, rolling update, and BlueGreen. The *in-place update* strategy first stops the old version instances then deploys the new version on the same instances. The *rolling update* strategy enables the user to configure the number of instances to be in-place updated at a time. The *BlueGreen* strategy creates new instances, deploys the new version on created instances, redirect the client requests to the new version and removes the old instances.

In terms of flexibility, AWS CodeDeploy does not support customizing strategies, neither extending the supported cloud platforms to extra cloud providers.

Strategies involve elementary operations that manipulate microservices. To deal with the specificities of each microservice, the DevOps team can extend some operations such as the *build, install,* and *start* scripts. More precisely, AWS CodeDeploy models an update process as a workflow consisting of a set of events hooks: *build, download, beforeInstall, install, afterInstall, start, validate, stop, beforeBlockTraffic, blockTraffic, afterBlockTraffic, beforeAllowTraffic, allowTraffic, afterAllowTraffic.* The DevOps team is in charge of providing the scripts for the different hooks, if specific actions have to be processed upon these events.

In case of failures, AWS CodeDeploy enables the DevOps team to stop an update, and provides a rollback capability for the updates processed with the BlueGreen strategy only.

3.2.5 Push2cloud

The framework push2cloud relies on an architecture-based approach for updating multiple microservices deployed on one single PaaS site. Currently, push2cloud only supports one PaaS platform: Cloud Foundry.

As with the previous frameworks, push2cloud models an update process as a sequence of (reconfiguration) steps. However, update processes are not defined by the DevOps teams, they are determined by a chosen strategy. Indeed, the DevOps team only needs to provide the desired microservices architecture and specify a chosen strategy to process an update.

Push2cloud provides DevOps teams with the following strategies:

- Simple: deploys all desired microservices. The strategy is used for the initial deployment of the microservices application.
- Redeploy: removes all the current microservices and deploys the desired microservices.
- BlueGreen: deploys the new version of the microservices while the old versions running, after the new versions deployed, removes the old versions.

Strategy are internally defined as JavaScript code files. Thus, the workflow of reconfiguration steps involved by a strategy are programmed as sequences of JavaScript instructions that update one or several microservices on one PaaS site.

Automation level

We consider here the usage effort for updating the application described in Section 3.2.2 (updating M_1 on $site_1$ and $site_2$ with the *BlueGreen* strategy) through the push2cloud framework.

The managed application consists of two microservices (M_1 and M_2) on two sites ($site_1$ and $site_2$). The DevOps team first needs to describe the desired architecture of the application. Push2cloud models a desired architecture as the combination of three models, expressed through three manifest files: microservice manifests (attributes of each microservice), release manifest (the microservices which compose the application), and deployment manifest (deployment configuration of the PaaS site).

As previously said, push2cloud only updates microservices deployed on one PaaS site. To issue an update on $site_1$, the DevOps team first requests push2cloud to generate the desired architecture *A-site1* through

merging and normalizing the manifests ($M_1.json$, $M_2.json$, *app.json*, and $site_1.json$). Then, the DevOps team can issue the update command by specifying the desired architecture *A-site1* and the strategy *BlueGreen*. Updating $site_2$ also requires processing these two commands.

During the execution of an update, push2cloud first identifies the updated microservice (M_1), deploys the new version of M_1, verifies it running, redirects client requests to the new version, and removes the old version as specified in the strategy file *bluegreen.js*.

Flexibility

As said in Section 3.2.5, the DevOps team does not have to define the steps of an update process. This simplifies the work of the DevOps team to process an update. However, it also reduces the capacities of the DevOps team to control the update process.

For instance, the DevOps team cannot integrate customized tasks during an update process, such as human approval steps. Indeed, push2cloud automatically generates the update process from the given strategy and target architecture, and runs the entire process from the beginning to the end.

To customize an update process, DevOps team may program their proper strategy in JavaScript. Elementary reconfiguration operations wrapping Cloud Foundry operations (e.g., create, upload, compile and start microservice etc) are made available. However, push2cloud does not provide any comprehensive documentation for this.

In terms of cloud platforms, push2cloud only supports the Cloud Foundry PaaS, but it claims to be able to be extended to support other platforms. Nevertheless, it is not an easy work. First, the microservices architectural model defined by push2cloud is based on the Cloud Foundry microservice model. The model may not be easily adaptable to other PaaS solutions, as each PaaS solution defines its proper attributes. Second, all the strategies needs to be re-implemented for other PaaS solutions, as the current implementation of strategies is based on specific PaaS operations. As it can be difficult to adapt these operations to other PaaS, the strategies implementations can be considered as staying specific to each PaaS solution.

Reusability

The behavior of a push2cloud update process is defined by the chosen strategy. A push2cloud strategy is microservice-independent. Taking the previous example, in case of updating another microservice M_2, the DevOps team can directly reuse the BlueGreen strategy. In addition, a push2cloud strategy is independent of the microservices application architecture.

However, push2cloud only allow to update microservices deployed on a single PaaS site. Therefore, the management of updates across sites (e.g.,

migrating between two sites) is left to the DevOps team.

Robustness

Push2cloud provides configurable parameters (timeout, retry, grace period) for all its provided operations. The parameter *grace period* refers to the duration for checking a microservice health after it is started. The push2cloud identifies an application as correctly deployed if no failures are detected during the grace period.

When an update process is failed in the middle of the execution, the microservices application is left into an intermediate state. Some strategies (*Simple* and *BlueGreen*) cannot update the microservices from an intermediate state. In such cases, the DevOps team has three options:

- manually resumes the update process by directly using elementary Cloud Foundry commands.
- manually rolls back to the previous state by removing the failed microservices through executing dedicated Cloud Foundry commands, then re-issues the push2cloud update.
- re-issues the push2cloud update with the strategy *Redeploy*. The strategy *Redeploy* can update the microservices from any intermediate state. However, it imposes microservice downtime because it stops and undeploy all microservices before redeploying them.

3.2.6 Other related approaches for managing Microservices

Beside industrial frameworks, there are also a few on-going research works related to microservices and their management.

Especially, [64] automates the configuration and the deployment of microservices. The proposed framework calculates an optimal target architecture satisfying user requirements and minimizing the number of used virtual machines. Then, it generates the sequence of deployment actions for automating the deployment of this target architecture on a IaaS provider. Compared to the issue addressed in this thesis, this framework only manages the initial deployment of microservices on single IaaS provider (i.e., OpenStack). The case of updating an existing deployment is not considered.

Inspired by [64], [65] proposes a similar approach which automates the update of microservices deployment through a goal-oriented approach. The proposed framework generates an optimal target architecture, then proceeds to the deployment if the target is agreed by the DevOps team. A main limitation of the approach is that it implies microservices to be programmed in the particular language Jolie [66], and deployed on a particular IaaS provider (i.e., Amazon AWS EC2). Although its deployment principle can

be extended to other microservices programming languages and cloud platforms, the complexity of these varieties (i.e., deploying microservices written in various languages and using heterogeneous operations provided by various cloud solutions) is not considered. In addition, the proposed framework does not consider update strategies. Therefore, its deployment process is simply to remove old versions microservices, install new versions, and configure the dependencies between deployed microservices.

Other works, [67, 68, 69] have introduced formalisms for automating deployment processes, but they consider components having dependent lifecycles and focus on the management of dependencies.

Finally, [70] investigates the problem of synthesizing a plan for deploying a target architecture for cloud components, considering capacity constraints and conflicts. It formalizes the problem of deploying a cloud application as the problem of planning a sequence of actions reaching a given target architecture. This work only considers the initial deployment of a cloud application.

3.2.7 Summary

Update process All the previously considered frameworks aim at simplifying the processing of microservices updates. Spinnaker and UCD manage updates on multi-clouds, while AWS CodeDeploy and push2cloud only manage mono-site updates. In addition, the pipeline of AWS CodeDeploy only updates a single microservice.

Both Spinnaker and UCD provide an imperative usage for the DevOps team. That is, the DevOps team defines how to process each updates, in terms of pipelines composed of a sequence of reconfiguration steps. Predefined set of step templates are provided . AWS CodeDeploy adopts a quite different approach, more event-based, through providing predefined update pipelines associated to customizable hooks. Push2cloud provides a declarative approach, allowing the DevOps team to only describe the target architecture (multi-microservices but mono-site) and a strategy for updating the application.

Automation level Each framework provides an interface exposing update operations that work at a particular level of abstraction. The DevOps team uses this interface to define specific update processes.

Figure 3.6 illustrates some different levels of interfaces. Figure 3.6(a) shows an interface that is at the level of elementary PaaS operations, allowing to manipulate a single attribute of a microservice on one PaaS site. The interface shown in Figure 3.6(b) allows to manipulate a microservice as a whole, automating the update of multiple attributes. The next level interface targets multiple microservices on one PaaS site (shown in Figure 3.6(c)) or multi-microservices on multi-sites (shown in Figure 3.6(d)).

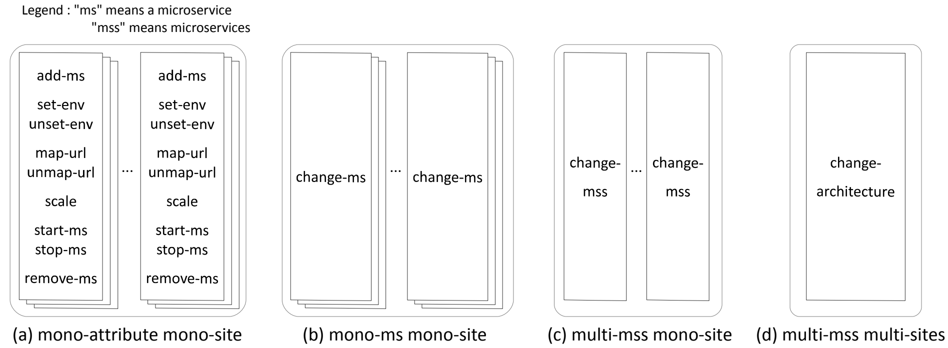


Figure 3.6: Example of framework interfaces with different automation levels (different target entities)

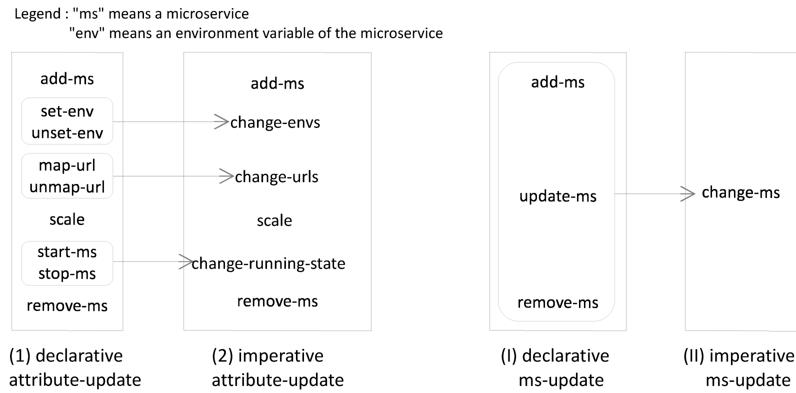


Figure 3.7: Framework interfaces with different automation levels

Flexibility The DevOps team needs not only automation to easily deliver updates, but also flexibility to satisfy specific needs.

To extend the supported cloud platforms, Spinnaker and UCD require to program a new version of the reconfiguration steps according to the newly considered PaaS. Push2cloud does not provide native support for extending the set of supported PaaS.

To extend the supported strategies, Spinnaker allows the DevOps team to customize strategies by defining new scripts composed of reconfiguration steps. UCD also allows customizing strategies, through the concept of process template. In push2cloud, customizing a strategy requires to program the according update workflow based on elementary PaaS operations. AWS CodeDeploy does not support extending the set of supported cloud platforms and strategies.

Reusability In Spinnaker, the user-defined pipelines are application-dependent and PaaS-dependent. Thanks to the immutable principle, the defined pipelines can be adaptable to the changes of various attributes.

However, the architectural and topological changes requires to construct specific pipelines. UCD makes the user-defined pipelines easier to be reused through the feature of process template. However, it disallows to reuse the pipelines across PaaS solutions. The update process defined in AWS CodeDeploy is application-specific. As a declarative approach, the update process of push2cloud is application-independent and change-independent. To reuse the update process on other PaaS solutions is still challenging for the moment.

Robustness Regarding failures, all the considered frameworks integrate a retry capability to fix transient failures, such as a short-lived network disconnection. Such failures are thus quite transparently managed for the DevOps team. Non-transient failures, such as hardware or software crashes (e.g., machine crash or microservice crash), are harder to manage. When such failures interrupt an update process, the microservices are left into an intermediate state. It should be noted here that script-based approaches are usually not idempotent (except Spinnaker that adopts an immutable approach), which prevents restarting an interrupted script after a failure. On the contrary, failures often require either to rollback-restart the entire update process, or to analyze the current state of the microservices to determine how to continue. Thus, for non-transient failures, the DevOps team should either manually continue the update process, or rollback to a previous stable state and rerun the pipeline. Only Spinnaker, UCD and AWS CodeDeploy automate the rollback process.

Conclusion Table 3.1 summarizes the considered update frameworks in terms of the update process model, the automation level, the flexibility, the reusability, and the robustness aspects.

To conclude, in terms of automation level and reusability, the architecture-based approach proposed by push2cloud requires the less usage effort for the DevOps team, compared to imperative approaches used by Spinnaker, UCD, and AWS CodeDeploy. However, push2cloud does not fully exploit the potential of an architecture-based approach, mainly because update processes are not defined at an architectural level. They are defined as sequences of JavaScript instructions that do not follow any particular programming model. Accordingly, push2cloud provides DevOps teams with a low level of control or observability over update processes.

Apart from push2cloud, most existing frameworks adopt a script-based approach to update a microservice application, where DevOps teams specify a pipeline of low-level operations to execute. The main limitation is certainly that script-based approaches are imperative, leading to the following limitations or constraints:

- First, DevOps teams often have to compose the scripts, which faces

Table 3.1: Summary of automation frameworks

	Spinnaker	**UCD**
Update Model	user-defined process composed of pre-defined steps	user-defined process composed of pre-defined steps
Auto level	⋆⋆ imperative; mono-attribute mono-site	⋆⋆ imperative; mono-attribute mono-site
Flexibility	customizable workflow pre-defined steps difficult to extend PaaS partial support to extend strategies	customizable workflow customizable steps support to extend PaaS support to extend strategies
Reusability	application-specific PaaS-specific	application-specific microservice-indep. templates PaaS-specific
Robustness	retry; rollback-restart	retry; rollback-restart
	AWS CodeDeploy	**push2cloud**
Update Model	pre-defined process composed of user-defined steps	process delivering user-defined target architecture
Auto level	⋆ mono-microservice mono-site	⋆ ⋆ ⋆⋆ declarative; multi-microservices mono-site
Flexibility	pre-defined workflow customizable steps disallow to extend PaaS disallow to extend strategies	pre-defined workflow pre-defined steps difficult to extend PaaS support to extend strategies
Reusability	application-specific platform-specific	application-independent PaaS-specific
Robustness	retry; rollback-restart	retry

the usual coding and debugging challenges.

- Second, they need to check that scripts are compatible with the current state of the microservices to update.
- 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 [5], 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.

[5]To gain such rollback capability without strongly increasing the complexity of scripts, Spinnaker adopts an immutable approach. However, this induces unavoidable additional resources cost at update time, preventing using any update strategy delivering a zero-cost property.

Chapter 4

Proposition

Contents

4.1 Usage Principles 36
4.2 Architectural Model 38
4.2.1 Data-Structure 38
4.2.2 Elementary Operations 39
4.2.3 Introspection and Reconfiguration of a Microservice Application 41
4.3 Strategy-driven Updates 45
4.3.1 Update Process Overview 45
4.3.2 Strategy-driven Update Protocol 46
4.4 Strategy Programming 49
4.4.1 Strategy Design 49
4.4.2 Didactic case: the BlueGreen Strategy 51
4.5 Update Robustness 54
4.5.1 Core principles 54
4.5.2 Identification of faults 56
4.5.3 Summary 61

This chapter presents our proposition: an automation framework named DMU (Declarative Microservices Update), allowing to update microservices running on heterogeneous PaaS platforms, in an automatic manner. This framework allows DevOps teams to launch *update processes* evolving a set of microservices from their current architecture toward a desired target architecture given as input. The architectural changes supported by our DMU framework include usual architectural changes, typically: code version, environment variables, instance number of one or multiple microservices (detailed in Section 2.5).

The essential property of the DMU framework is that it integrates the concept of *strategies*, aiming at taking care of the SLA properties of the

microservices during the overall update process. Most microservices are indeed subject to non-functional requirements (i.e., a specific SLA trade-off) in terms of availability, performance and resource consumption and it is important to keep these properties during an update session.

Concretely, a *strategy* determines the path of intermediate architectures that will be followed to update microservices while maintaining some non-functional requirements. For instance, considering an update migrating a set of microservices from site S_a to site S_b, a first strategy, focusing on minimizing resource consumption, may remove the microservices on site S_a before recreating them on site S_b. Another strategy, favoring the service availability, may create new microservice instances on site S_b before removing old instances on site S_a.

Overall, the proposed DMU framework allows a DevOps team to update microservices through simply giving as input: 1) a desired *target architecture* and 2) a chosen *strategy*. The *target architecture* defines how microservices instances are to be spread over the PaaS sites and what are their expected configuration (attributes values). The strategy determines how to reach the target architecture such as to preserve some SLA properties.

In the following of this chapter, we will describe in details how the DMU framework supports such strategy-driven updates. The organization is as follows. Section 4.1 presents the usage principles of the framework. Section 4.2 presents the architectural model of the DMU framework, used to express a target architecture for a microservice application to update. Section 4.3 defines the concept of strategy, and describes an update protocol integrating such concept. Section 4.4 presents the strategy programming model. Finally, Section 4.5 addresses the robustness aspect, explaining how the DMU framework handles faults that may occur at update time.

4.1 Usage Principles

Three main kinds of users may interact with the DMU framework, reflecting the three main roles involved: *the microservice manager, the strategy programmer* and the *PaaS connector programmer*.

- The *microservice manager* refers to the DevOps team that uses the DMU framework to deploy and update microservices on a set of distributed PaaS platforms. For any microservice to update, the microservice manager specifies the target architecture along with the chosen strategy. In the following of this document, we may use indifferently the terms *microservice manager* or *DevOps team*.

- The *strategy programmer* refers to the developer of update strategies. The DMU framework provides a set of predefined update strategies, that can be extended over time and/or specialized for the account of dedicated applications. New strategies may be defined to take into account new SLA constraints, or to deal with the specificities of a particular microservice application. Strategies are programmed as Java classes following some particular programming patterns, that will be described in the following of this chapter.

- The *PaaS connector programmer* refers to the person in charge of integrating new PaaS platforms as back-ends of the DMU framework, in order to allow updating microservices running on heterogeneous PaaS platforms. As for strategies, the DMU framework supports a predefined set of PaaS platforms (Cloud Foundry, Heroku), that can be extended over time. Integrating a new PaaS platform goes through defining the mapping between the PaaS specific commands and a generic canonical PaaS interface that is described in the following of this chapter.

Figure 4.1 schematizes these roles. The right side of the figure illustrates distributed PaaS platforms hosting microservices that the microservice manager wish to deploy or update with the DMU framework.

From an operational point of view, the microservice manager that wants to update a microservice application should firstly launch the DMU framework. This can be done on any host as the DMU framework is a standalone execution unit. The only constraint to care about is that the DMU framework requires a network connection towards the PaaS sites hosting the microservices to update.

Through the DMU framework, the microservices manager can invoke two main commands:

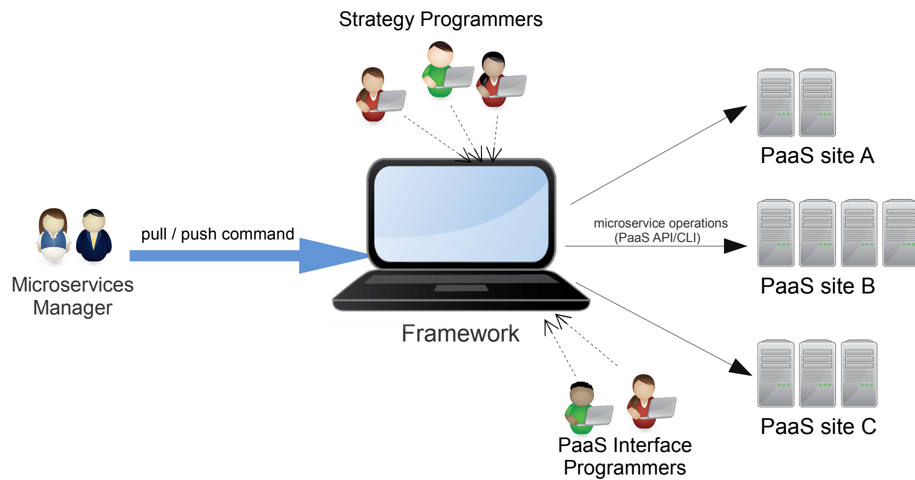


Figure 4.1: Involved roles around the DMU framework (operational view)

- a *pull* command allowing to get the current architecture of an application,
- a *push* command updating an application towards a given target architecture, following a given strategy.

Each time the *push* command is triggered, an *update process* is launched. The microservices manager may then follow the processing of this process, step by step, if desired. (S)he may observe the evolution of the architectural state of the microservices. This architectural state is expressed through a dedicated *microservice architecture model*, that is presented in the following section.

4.2 Architectural Model

This section defines the notion of *microservice architecture* that is central to the DMU framework. For simplicity, we simply use the term *architecture* in the following of the document. Section 4.2.1 describes how we modelise an *architecture* in terms of data-structure. Then, Section 4.2.2 presents the elementary operations allowing to manipulate an *architecture* data-structure. Finally, Section 4.2.3 proposes a basic protocol relying on these elementary operations, allowing a DevOps team to introspect and/or reconfigure a microservice application.

4.2.1 Data-Structure

Listing 4.1 shows the data structure that models the architecture of a microservices application. Overall, this data-structure specifies how microservices are deployed on PaaS sites and how they are configured (Listing 4.1[1]).

More precisely, in the given data-structure, the microservices belongs to an application whose identifier is app_{id} (we assume that any application is uniquely identified). Each *site architecture* corresponds to a PaaS site, uniquely identified as well ($site_{id}$). A *site architecture* specifies how to access the site and which microservices are deployed on the site. Typically, the site $accessInfo$ contains the *url* of site endpoint and the user credentials for accessing the site.

```
Architecture = (String app_id, Set<Site>);
Site = (String site_id, Set<String attr, String val> accessInfo, Set<Microservice>);
Microservice = (String ms_id, Set<String attr, String val>);
```

Listing 4.1: Architecture model

A *microservice architecture* describes a given microservice deployed on a given PaaS site. We assume that any microservice is uniquely identified (ms_{id} identifier). We also assume that the current architectural state of such microservice can be expressed as a set of (attribute, value) pairs, including both PaaS-common and PaaS-specific configuration attributes. PaaS-common attributes include attributes that we identified as being common to existing PaaS platforms [36, 35, 34, 32, 40]. These common attributes mainly correspond to the following ones:

1. *name* - symbolic name of the microservice.
2. *code* - link to the place where the code [2] of the microservice can be found.

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

[2] The code here means not necessarily the source code of the microservice. It can be any package format required to be uploaded to PaaS for deploying the microservice. For example, in case of a microservice on Kubernetes, the attribute describes the container image of the microservice.

3. *state* - lifecycle state for the microservice:
    - *CREATED*: the microservice is created.
    - *RUNNING*: the microservice is running.
    - *FAILED*: the microservice is failed.
4. *nbInstances* - number of instances of the microservice.

Figure 4.2 illustrates an *application architecture* composed of two microservices (m_1 and m_2) deployed both on two heterogeneous sites ($site_1$ and $site_2$) and a third microservice m_3 only deployed on $site_2$. Notice that the architecture of the microservice m_1 is different across different sites (e.g., $site_1$ and $site_2$) in terms of the attribute value (e.g., the value of *instances*) and the attributes definition (e.g., *memory* on $site_2$).

Figure 4.2: An example of microservices architecture

As previously said, the set of key-value pairs describing a microservice is extensible. Especially, each PaaS solution can extend pre-defined attributes with additional ones in the model. For example, only the PaaS solution of $site_2$ in the example requires the attribute *memory* which specifies the memory quota of each microservice instance, while $site_1$ does not require this information.

A last important point to precise regarding the data-structure modeling a microservice is that this data-structure is used both to portray the current and the target architectures of a microservice.

4.2.2 Elementary Operations

To manipulate microservices, the DMU framework relies on the operations provided by PaaS sites (through their API and/or CLI). Although different

PaaS platforms provide syntactically different operations, they mostly share a common semantics. We abstract this semantics through a set of canonical *PaaSOperations* allowing to manipulate a microservice architecture. Following a CRUD (Create, Read, Update, Delete) approach [71], this canonical (Listing 4.2) allows to add, get, modify and remove microservices .

Specifically, the canonical operations behave as follows:

- *get*: returns the architecture of the microservices currently deployed on the given PaaS site and associated to the application app_{id}.
- *add*: deploys a new microservice for the account of the application app_{id}.
- *remove*: deletes an existing microservice for the account of the application app_{id}.
- $modify$: evolves the attributes of a microservice deployed for the account of the application app_{id}. This operation takes as input the current and target attributes of the microservice to modify.

For each PaaS platform managed by the DMU framework, the PaaS-specific implementation of this canonical interface (called *PaaS connector*) should be defined. Currently, we provide the implementations of the *PaaS connector* for the Cloud Foundry, Heroku, and Kubernetes PaaS platforms in the open-source prototype of the DMU framework [72]. As an illustration, we explain here the implementation of Kubernetes and Cloud Foundry PaaS connectors.

Regarding the PaaS Kubernetes, the $PaaSconnector$ interface is in fact directly mapped on Kubernetes configuration commands: *create*, *delete*, *replace* and *get*, such commands being based on a CRUD approach as our canonical interface.

Regarding the PaaS Cloud Foundry, it is a bit less direct. For the canonical operations *get*, *add*, and *remove*, Cloud Foundry provides corresponding operations. The operation $modify$ is a bit more complex to implement

```
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 app_id, Microservice m, Microservice m_next); a
}
```

Listing 4.2: Architecture manipulation (PaaSOperation interface)

[a] To ensure m and m_{next} describe the same microservice, $m.id$ is enforced to be equal to $m_{next}.id$ at the beginning of the function.

because Cloud Foundry provides distinct operations to update specific attribute(s). Therefore, depending on the change to apply on the microservice, the operation $modify$ requires different Cloud Foundry operations workflows. The following pseudo-code (Listing 4.3) shows the implementation of the $modify$ operation. By comparing the attributes' value between the current (m) and desired (m_{next}) microservice architecture, it determines the sequence of Cloud Foundry operations to process.

```
CloudFoundryConnector.modify(m, m_next) {
    Workflow modifyWF = new SerialWorkflow();
    IF (dif(m, m_next, code)):
        modifyWF.addStep(uploadCode(m.id, m_next.code));
        m.code = m_next.code;
        m.state = UPLOADED;
    boolean needRecompile = false;
    IF (dif(m, m_next, env)):
        modifyWF.addStep(updateEnv(m, m_next));
        m.env = m_next.env;
    IF (dif(m, m_next, services)):
        modifyWF.addStep(updateServices(m, m_next));
        m.services = m_next.services;
    IF (needRecompile AND isCompiled(m)):
        modifyWF.addStep(recompile(m));
        m.state = STAGING;
    Set<Route> addedRoutes = m_next.routes \ m.routes;
    IF (addedRoutes NOT EMPTY):
        modifyWF.addStep(addRoutes(addedRoutes));
        m.routes = m.routes ∪ addedRoutes;
    Set<Route> removedRoutes = m.routes \ m_next.routes;
    IF (removedRoutes NOT EMPTY):
        modifyWF.addStep(removeRoutes(removedRoutes));
        m.routes = m.routes \ removedRoutes;
    IF (dif(m, m_next, nbInstances)):
        modifyWF.addStep(scaleNbInstances(m, m_next));
        m.nbInstances = m_next.nbInstances;
    IF (dif(m, m_next, state)):
        modifyWF.addStep(updateStateAutomaton(m, m_next));
        m.state = m_next.state;
    return modifyWF;
}
```

Listing 4.3: $CloudFoundry$ $modify$ operation

4.2.3 Introspection and Reconfiguration of a Microservice Application

Relying on the *PaaSOperations* interface, the DMU framework defines two main introspection and reconfiguration functions:

- $introspect(app_{id}, List < Site >)$: retrieve the current architecture of a given microservice application deployed on the given PaaS sites.
- $reconfigure(app_{id}, arch)$: reconfigure a given microservice application towards the given architecture ($arch$). This *reconfigure* function is

strategy-unaware, meaning that it reconfigures the application towards the given architecture *arch* through the most direct path, without trying to protect any SLA properties.

The function *introspect* is implemented as shown in Listing 4.4. It retrieves the currently existing microservices on each site by calling the *get* function of the *PaaSOperations* interface.

```
introspect(app_id, List<Site> sites) {
    // create an empty architecture model
    Architecture Acurrent = new Architecture();
    // get the microservices on each site
    for (PaaSSiteAccess site : sites) {
        // get the PaaSOperation implementation class based on the type of site
        PaaSOperations op = Class.forName(site.type).newInstance(site);
        // retrieve the microservices currently managed by a site
        List<Microservice> microservices = op.get(app_id);
        Acurrent.add(site, microservices);
    }
    return Acurrent;
}
```

Listing 4.4: Core introspect protocol

The function $reconfigure$ is defined as shown in Listing 4.5 [3]. For each site, the $reconfigure$ function firstly computes an architectural diff [54] between the target and current architecture, determining the set of microservices to reconfigure (i.e. added, removed, or modified). For each microservice to reconfigure, the corresponding canonical operation (from the interface PaaSOperation) is added to a per-site workflow. These workflows are finally processed, on all sites in parallel.

Any canonical operation that is processed on a given PaaS site is translated into its PaaS-specific version. For instance, let's consider the usecase illustrated by(Figure 4.3). To upgrade the microservice m to the next code version (modeled as m') deployed on two PaaS sites S_a (Cloud Foundry) and S_b (Heroku), the $reconfigure$ function behaves as follows:

- First, for the site S_a, it compares the microservices on S_a in the model of the target architecture (i.e., m') and the current architecture (i.e., m); it finds that *changeSet* contains only a *modified* microservice.
- Since *changeSet* does not contain any *added* or *removed* microservices, it only adds the operation $CloudFoundry.modify(m, m')$.
- The previous steps are executed similarly for S_b, adding the operation $Heroku.modify(m, m')$.

[3] Notice that, when processing a *get* operation, the concerned PaaS may be processing another operation. For consistency purposes, a PaaS connector implementation should ensure to get a stable snapshot of the microservice architecture (i.e., wait for in-processing operations to finish).

```
reconfigure(Architecture Acurrent, Architecture Atarget) {
    // create an empty parallel workflow to store the reconfiguration workflow
    Workflow reconfigure = new ParallelWorkflow();
    // get the reconfiguration workflow for each site
    for (PaaSSiteAccess site : desiredArchitecture.listPaaSSites()) {
        // create an empty parallel workflow to store the workflow for reconfiguring the site
        Workflow reconfigSite = new ParallelWorkflow();
        // compare the microservices on the site in current and target architecture
        ChangeSet changeSet = diff(Acurrent[site], Atarget[site]);
        // get the PaaSOperation implementation class based on the type of site
        PaaSOperations op = Class.forName(site.type).newInstance(site);
        // call the add operation for all added microservices
        for (Microservice addedMs : changeSet.addedMs()) {
            reconfigSite.addStep(op.add(addedMs));
        }
        // call the remove operation for all removed microservices
        for (Microservice removedMs : changeSet.removedMs()) {
            reconfigSite.addStep(op.remove(removedMs));
        }
        // call the modify operation for all modified microservices
        for (Entry<Microservice, Microservice> modifiedMs : changeSet.modifiedMs()) {
            reconfigSite.addStep(op.modify(modifiedMs.getKey(), modifiedMs.getValue()));
        }
        // add the site reconfiguration workflow to the entire reconfiguration workflow
        reconfigure.addStep(reconfigSite);
    }
    // execute the reconfiguration workflow
    reconfigure.exec();
}
```

Listing 4.5: Core reconfigure protocol

- These two operations are added to the workflow $reconfigure$.
- During the execution of the workflow, the CloudFloundry and Heroku connectors respectively invoke the sequence of PaaS specific operations corresponding to the $modify$ operation.
- Finally, the $reconfigure$ function waits for the result of these PaaS operations and returns.

Thus, so far, the DMU framework provides an *introspect* and a *reconfigure* operations. One can ask why updating a microservice application is not just *reconfiguring* the application towards a desired target architectures, as shown in Listing 4.6. We call such update as a strategy-less update (shown in Figure 4.4). The reply is the following: reconfiguring a microservice application relies on processing *add, remove* and *modify* canonical operations on the concerned microservices. Regarding the *modify* operation that allows to change the configuration of a running microservice m, most PaaS, including Cloud Foundry and Heroku, imply to follow the following three main steps:

- (1) stopping m (m should indeed be stopped before updating its code),
- (2) uploading the code of m',

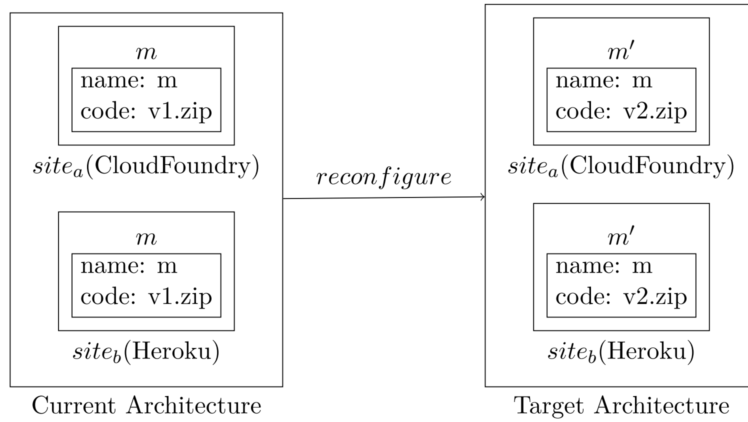


Figure 4.3: An example $reconfigure$ for upgrading microservice code

- (3) starting m'.

Such processing, taking several minutes in average (because of the code downloading), introduces a significant downtime for the updated microservice m. To upgrade m toward m' without downtime, the proposed way consists in using strategies (presented in section Section 4.3).

```
push(Architecture Atarget) {
    // retriving current architecture of the app app_id
    Architecture Acurrent = introspect(Atarget.app_id, Atarget.sites);
    // updating the app app_id towards Atarget
    reconfigure(Acurrent, Atarget);
}
```

Listing 4.6: Strategy-less update protocol

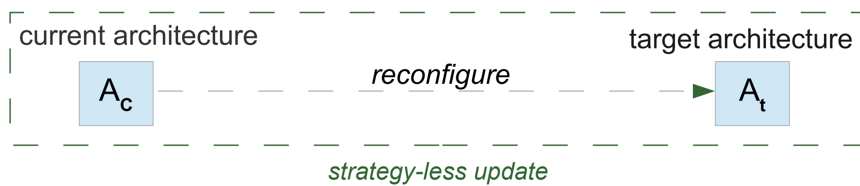


Figure 4.4: The Overview of Strategy-less Update (architecture-based)

4.3 Strategy-driven Updates

This section presents how the DMU framework takes into account the need to protect SLA properties when performing an update, through the concept of strategy. It is organized as follows. First, Section 4.3.1 introduces the principles of ”strategy-driven” updates. Then, Section 4.3.2 presents the implementation of an update protocol taking into account strategies.

4.3.1 Update Process Overview

A strategy aims at allowing to update microservice applications while protecting their SLA properties throughout the update process, as illustrated in Figure 4.5.

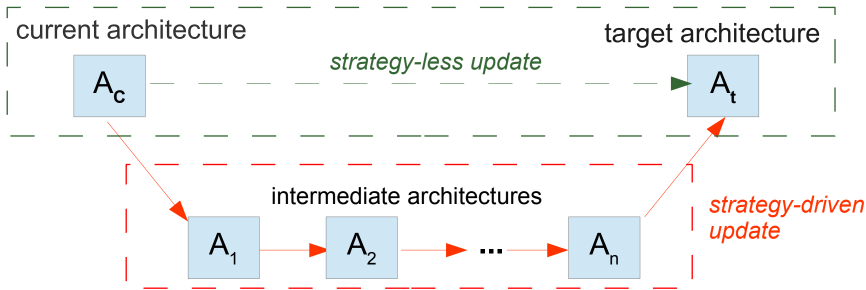


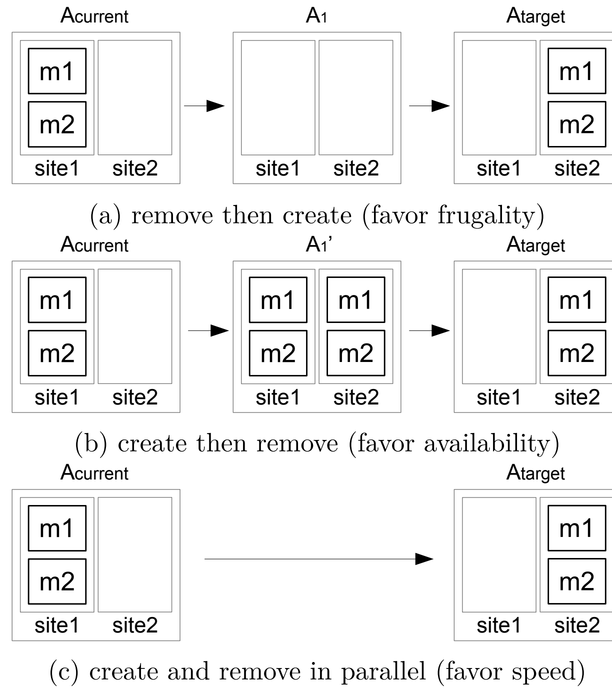
Figure 4.5: Update Process Overview

To this end, a *strategy* determines a path of *intermediate architectures* that will be followed to reach the target architecture while protecting some SLA properties (e.g., availability, performances, resource usage). Let’s consider that the strategy computes a path $I_0, I_1, .., I_n$ of intermediate architectures, the last intermediate architecture I_n matching the target architecture. The DMU framework will then reconfigure the microservices in n steps, reaching the next intermediate architecture at each step.

In order to give a concrete example of a strategy-driven update process, let’s consider a previously mentioned update case consisting of migrating some microservices from a PaaS site to another. Figure 4.6 shows the different paths of intermediate architectures corresponding to various strategies that can be used to process such update. In this figure, large boxes show PaaS sites (named $site_1$ and $site_2$), and the microservices deployed on them.

- Using a *RemoveAdd* strategy (Figure 4.6a): This strategy minimizes resource consumption. The microservices are firstly removed to get to the intermediate architecture A_1, then deployed on the new site to arrive at the target architecture.

- Using a *AddRemove* strategy (Figure 4.6b): This strategy favors service availability. Until the microservices arrives at the architecture A_1'



(a) remove then create (favor frugality)

(b) create then remove (favor availability)

(c) create and remove in parallel (favor speed)

Figure 4.6: The example of migrating microservices m_1 and m_2 from $site_1$ to $site_2$ with three strategies

where the microservices are co-running on both two sites, they are not removed from the old site.

- Using a *Straight* strategy (Figure 4.6c): Besides the two presented strategies, if the DevOps team prefers to finish the migration as soon as possible, and cares less about the consumed resource and the imposed downtime during the update, it can choose a *Straight* strategy to deploy microservices on the new site while removing them on the old site in parallel.

Notice that the DMU framework allows to launch an update process in a preview mode, simulating an update without applying it to the real system. The preview mode can help the DevOps team to decide between different strategies. The strategy programmer can also use the preview mode to debug and test the strategy.

4.3.2 Strategy-driven Update Protocol

With the previously presented *introspect* and *reconfigure* functions, the DMU framework evolves a given application toward a given target in the most direct way, without going through intermediate architectures. In other

words, with only *introspect* and *reconfigure* functions, only strategy-less updates can be processed by the DMU framework.

Taking into account strategies goes through providing a dedicated concept allowing to define them. Concretely, we modelize a strategy as a programmable unit providing one main feature: computing the next intermediate architecture to reach in order to update a microservice application towards a given target. As we place our framework in an object-oriented environment, a strategy is naturally mapped on a class defining a *next* method:

- *Strategy.next(Acurrent, Atarget)*: computes the next intermediate architecture to reach based on the current architecture of the application, the desired target architecture (A_{target})

To evolve from the current architecture to a desired target one –while conforming to a given strategy–, the DMU framework invokes *next* and *reconfigure* sequentially and iteratively (Listing 4.7 [4]). In other words, the update protocol is now defined as a *fix-point* parameterized by a strategy. At each step, the fix-point requests the strategy to compute the next architecture to reach, then reconfigures the microservices towards this next architecture, and repeat these two steps until reaching the target architecture.

```
push(Architecture Atarget, Strategy strategy) {
    Architecture Acurrent, Anext;
    Acurrent = introspect(Atarget.app_id, Atarget.sites);
    // fix-point, updating app_id towards Atarget
    while (Acurrent.differ(Atarget)) {
        // compute next intermediate architecture Anext
        Anext = next(Acurrent, Atarget, strategy);
        if (Anext == null) exit("target unreachable"); **
        // to reach the next architecture Anext
        reconfigure(Acurrent, Anext);
        Acurrent = introspect(app_id, Atarget.sites);
    }
}
```

Listing 4.7: Strategy-driven update protocol (fix-point)

There is one main reason why the update protocol is defined as a fix-point instead of calculating the whole sequence of intermediate architectures at the beginning. This reason is to allow the DevOps team to modify the strategy and/or the target architecture at each step. This feature may be especially useful when considering failures occurring at update time (see Section 4.5).

Similarly to the update protocol, the preview protocol (Listing 4.8) also invokes the *next* function iteratively for computing the next architecture for

[4] The line tagged with ** represents an error case that is discussed in Section 4.4.1.

arriving at the final architecture A_{target}. However, the preview protocol accumulates the sequence of architectures (*archSequence*) instead of invoking the *reconfigure* function, so that effective microservices are not impacted. The returned *archSequence* allows to visualize the path of intermediate architectures that will be followed when processing the update.

```
List<Architecture> preview(Architecture Ainitial, Architecture Atarget, Strategy
    strategy) {
   Architecture Acurrent, Anext;
   List<Architecture> archSequence;
   Acurrent = Ainitial;
   archSequence.add(Acurrent);
   // fix-point, calculating intermediate architectures towards Atarget
   while (Acurrent.differ(Atarget)) {
      // compute next intermediate architecture Anext
      Anext = strategy.next(Acurrent, Atarget);
      if (Anext == null) exit("target unreachable"); **
      // add the Anext into the sequence of architectures
      archSequence.add(Anext);
      Acurrent = Anext;
   }
   return archSequence;
}
```

Listing 4.8: Strategy preview protocol

Regarding the termination, the update fix-point (Listing 4.7) stops either when the target architecture has been reached, or when the strategy has no more changes to process. In case the target has not been reached [5], the DevOps team may simply restart an update with an adapted strategy and/or target architecture (explained in Section 4.5).

In the case where a strategy is badly programmed, especially when transitions generate opposite changes on the architecture, the fix-point may never terminate. To manage such case, the DevOps team can pre-check the termination of an update by using the *preview* mode.

[5] This case corresponds to the line tagged with ** in Listing 4.7.

4.4 Strategy Programming

Section 4.4.1 presents the programming model allowing to program strategies. We used this programming model to provide a set of predefined strategies with the proposed DMU framework, some of these strategies being presented in this chapter as a didactic case. In particular, Section 4.4.2 illustrates the strategy programming model through the well-known *BlueGreen* strategy.

4.4.1 Strategy Design

A strategy provides one main feature: computing the next architecture to reach according to the current and target architectures.

To achieve this, the DMU framework naturally models a strategy as an ordered list of *transition* elements (described in Listing 4.9). A transition manages an elementary update that may participate in an overall update process. To compute the next intermediate architecture, a strategy processes its transitions incrementally, until finding one having changes to apply on the current architecture.

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

    // compute the next intermediate architecture to reach
    Architecture next(Architecture Acurrent, Architecture Atarget) {
        // process transitions until finding one having changes
        // to perform on the current architecture
        for each Transition tr in transitions {
            Architecture Anext = tr.process(Acurrent, Atarget);
            if (Anext != null) return Anext;
        }
        return null;
    }
}

interface Transition {
    // deliver a (not null) next architecture if the transition has changes to process
    Architecture process(Architecture Acurrent, Architecture Atarget);
}
```

Listing 4.9: Strategy model

Pragmatically, a transition provides a *process()* method that 1) determines if its elementary update is relevant to get closer (i.e., go forward) to the target and 2) delivers a next architecture in accordance. For instance, a transition *Tadd* managing the adding of microservices behaves as follows (Listing 4.10). Comparing the current and target architectures, it firstly determines if new microservices have to be deployed. If no, it simply returns null. If yes, it delivers a next architecture containing the current microservices plus the new microservices to deploy. Symetrically, a transi-

tion *Tremove* determines if there are microservices to undeploy. If yes, it delivers an architecture containing the current microservices minus those to undeploy.

```
class AddRemoveStrategy extends Strategy {
    // manages addings and removings of microservices,
    // processing addings then removings
    transitions = new List(Tadd, Tremove);
}

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

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

Listing 4.10: Implementation of the example strategy *AddRemove* and its transitions

Figure 4.7 illustrates using the *AddRemoveStrategy* (Listing 4.10) to update 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.

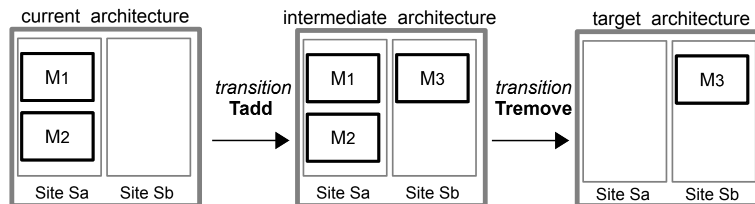


Figure 4.7: Update case with *AddRemove* strategy

- At the first step, the update fix-point processes the first transition ($Tadd$) of the strategy, that delivers the intermediate architecture (A)

composed of the current microservices plus M_3 deployed on S_b. The application is then reconfigured towards this intermediate architecture A.

- 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 one can define different strategies by changing the order of transitions. In the previous update case, the reverse sequence of transitions (i.e., $(Tremove, Tadd)$ instead of $(Tadd, Tremove)$) would generate a different path of intermediate architectures, removing M_1 and M_2 first, then adding M_3.

Notice also that transitions may apply changes over several steps of the fix-point. Let's consider a transition scaling up microservices horizontally (i.e., growing their number of instances) as follow. For each microservice to scale, new instances should be deployed and started one by one [6]. Each time it is processed, this transition returns a next architecture in which every microservice to scale has one more instance. When all microservices reach their target number of instances, it simply returns null.

4.4.2 Didactic case: the BlueGreen Strategy

As a representative case, we explain how to program the *BlueGreen* strategy that updates an application without downtime – through installing, starting and testing the new version (the green one) before uninstalling the current version (the blue one). Once the green environment is ready, incoming requests should be routed to it.

We define such strategy through a simple sequence composed of four transitions (Listing 4.11).

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

Listing 4.11: BlueGreen Strategy

These transitions perform the following changes.

- $Tadd$: deploys microservices defined in the target architecture but undefined in the current architecture.

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

- *Tupdate*: deploys the green version of the microservices that are updated 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 the previous step.
- *Tremove*: removes microservices entities that are undefined in the target architecture but defined in the current architecture.

The implementation of the transitions *Tadd* and *Tremove* are quite similar to the code given in Listing 4.10. We show hereafter the code of the transitions *Tupdate* (Listing 4.12) and *Tswitch* (Listing 4.13).

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

Listing 4.12: Implementation of the transition Tupdate

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

Listing 4.13: Implementation of the transition Tswitch

To illustrate how the DMU framework process an update with the implemented *BlueGreen* strategy, let's consider the case described in Figure 4.8.

To upgrade M_1 towards M_1' and remove M_3, the DevOps team makes the following request to the DMU framework: *push(A_t, BlueGreen)*.

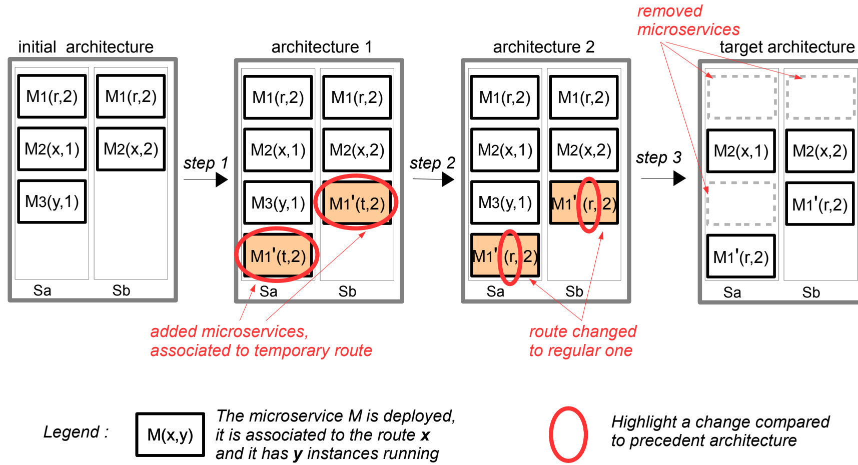


Figure 4.8: Processing the transitions of the BlueGreen strategy

The DMU framework starts by getting the current architecture (let's call it A_c) for the application A and enters the *update* fix-point. It processes the first transition $Tadd$, that has no changes to perform as A_t does not include additional microservices compared to A_c. It then processes the second transition ($Tupdate$) that delivers the *architecture 1* shown in Figure 4.8. In this architecture, M_1 is deployed both in its current version of code and in its new (green) version (M_1'). Then, the framework reconfigures the application towards *architecture 1* by calling the *add* operation of the two sites ($S_a$and S_b) in parallel. Notice that external client requests continue to be routed to M_1 so far, as M_1' is assigned a temporary route.

At the second step of the fix-point, transitions are processed again. The first and second transitions have no more changes to perform. The third transition ($Tswitch$) computes the *architecture 2* where M_1' is now associated to the regular route, and the old version of M_1 has been removed. The $reconfigure$ protocol calls the *modify* operation (changing the *route* of M_1') on the two sites to deliver *architecture 2*.

Once the application is reconfigured towards *architecture 2*, the framework restarts processing the transitions. Only the last transition ($Tremove$) has some changes to process, producing the *architecture 3*. The $reconfigure$ protocol calls the *remove* operation for M_1 and M_3 in parallel to deliver *architecture 3*. Finally, the *architecture 3* matches the target, leading to the completion of the update fix-point.

4.5 Update Robustness

In this section, we consider faults occurring during an update process. Section 4.5.1 presents the core principles underlying the management of such faults. Section 4.5.2 details various types of faults that may impact an update and how to handle such faults. Finally, Section 4.5.3 summarizes how the DMU framework helps the DevOps team to handle each type of faults.

4.5.1 Core principles

Automating the update of microservice applications requires to consider the faults that may occur during an update. Before explaining how we address faults, let's recall that nowadays, there are millions of DevOps teams that (manually) perform daily updates of their microservice applications, through the Command Line Interface provided by PaaS sites. To process such updates, DevOps teams send several commands to one or more PaaS sites, each command being entirely processed on a PaaS site before the next command is sent.

During such manual update processes, faults may occur: a microservice may fail to process some command, a PaaS site may crash, the network connecting a PaaS site to the host used by the DevOps team may face some disconnection. However, PaaS sites follow a fail-stop failure model [73] that enforces the consistency of the microservices they host. This means that once a PaaS site is up and reachable, the microservices it hosts can be introspected and reconfigured. Thus, if a PaaS site crashes during an update, it should be relaunched, but once this is done, the DevOps team can always issue the necessary CLI commands to pursue the update.

Going back to our framework, we do not provide a better level of robustness regarding faults, we just rely on the consistency property ensured by PaaS sites. In other words, as with PaaS sites, when using our framework, DevOps teams have the guarantee that their microservice applications stay consistent, thereby manageable. Nevertheless, if we do not provide a better level of robustness, the fact that we automatize updates allows to simplify the work of the DevOps team in the presence of faults. We will explain how this is achieved in the next sections.

Going one step further, it is important to understand that the consistency property provided by PaaS sites allows to leverage a **kill-restart** capability at the level of our framework. Indeed, whatever the moment when an update is interrupted, either due to a fault or due to a voluntarily interruption from the DevOps team, a new update process can always be started. Overall, the kill-restart capability relies on the following three main properties.

- *PaaS consistency.* Whatever the current architectural state for a microservice application, it is a regular state for the PaaS sites hosting

it, meaning that microservices can always be introspected and/or reconfigured.

- *Stateless framework.* The DMU framework does not keep any model@runtime for the managed microservices, a data-structure that is especially difficult to maintain in the presence of faults. The reason for the difficulty is that the data should stay consistent with the effective microservices architectures. Instead, the DMU framework dynamically gets the current architecture of microservices to update, using introspection capabilities provided by PaaS sites.

- *Update idempotence.* Following an architecture-based approach, transitions compare the current and target architectures to determine the changes to perform. Once a change has been applied, the DMU framework does not redo it again. Thus, restarting an update that was previously issued does not cause side effects.

Notice that in any case, the DevOps team is free to either pursue a failed update update towards the initial target through the initial strategy, or change the target and/or the strategy. Changing the target architecture offers a way to fix some microservice configuration. It also offers a way to roll-back an entire update, by defining the new target architecture as the initial architecture that the microservices had when launching the update. Figure 4.9 illustrates an update that fails, the DevOps team deciding to pursue the update with a new strategy S' and a new target architecture At'.

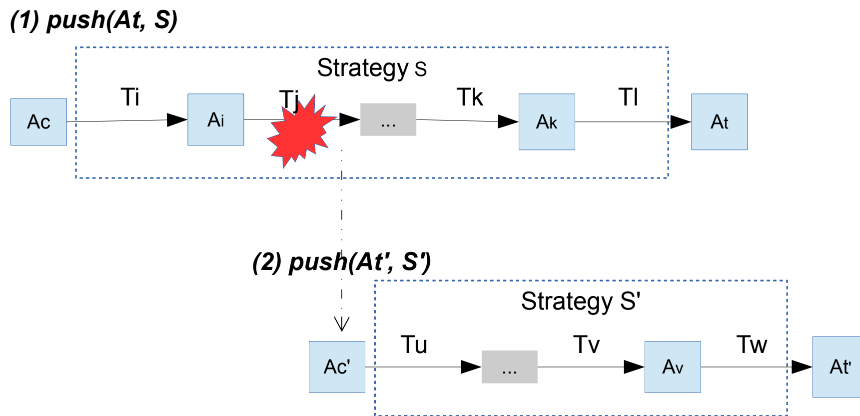


Figure 4.9: Management of failures at update time by adapting the strategy and the target

Upon a failure, the DevOps team may intend to pursue the previously interrupted update towards the given target. To this end, it may just reissue the initial *push* command if the used strategy is *versatile* (Figure 4.10a). We consider a strategy as versatile if it does not make assumptions on the initial architecture of the microservice application to update. That is, the

strategy accepts any architecture as the initial architecture of the update. A typical versatile strategy is the *Straight* one, that updates a microservice application towards a given target through the most direct path, without going through intermediate architectures.

Let's consider a fault occurring during the transition between the architecture A_i and the architecture A_{i+1}. If the used strategy is not versatile, the DevOps team may then issue a push command with the *Straight* strategy (a versatile one) and the next architecture (A_{i+1}) given as target (Figure 4.10b). [7] Once done, the DevOps team may then re-issue the initial *push* command, with the initial strategy and the initial target architecture given as input.

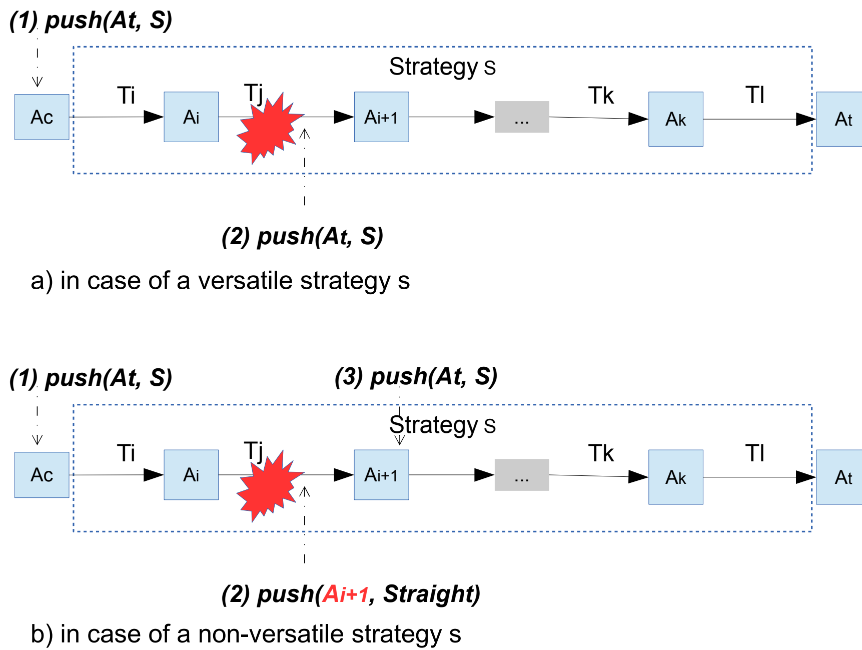


Figure 4.10: Management of failures at update time by pursuing the interrupted update

4.5.2 Identification of faults

Before classifying the faults that may occur during the update, we recall here the overview of the DMU framework (Figure 4.11). Three actors are involved during the update: the DevOps team, the DMU framework, and the PaaS sites. During the update, the DMU framework translates the DevOps team's commands into the operations sending to the PaaS sites which host the microservices.

Based on this operational architecture, and following the literature on

[7] This next architecture can be retrieved from the DMU framework execution log.

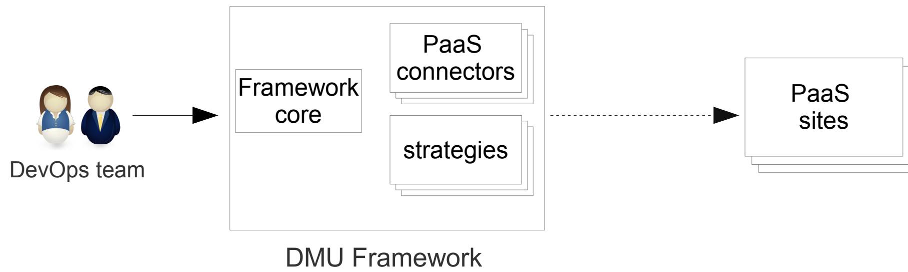


Figure 4.11: The operational architecture of the DMU Framework

faults [74], the faults to consider can be classified into three non-exclusive groups [8]:

- development faults: we consider here the software faults introduced during the development of the DMU framework (such as null dereferences), leading to either an interruption or an exception raising at execution time.
- physical faults: we consider here the faults originated in hardware (including the framework servers, the PaaS servers and network devices), such as electric power outage or hardware damage.
- interaction faults: we consider here the faults originated outside the DMU framework, including the user mistakes (e.g., an erroneous input) and faults propagated from the external components which interact with the DMU framework (e.g., a PaaS site failure).

All these kinds of faults force an update to be interrupted. When an update is interrupted due to faults, the following steps are required to recover the faults:

1. identify the fault origin
2. fix the faulty components (hardware and software)
3. repair the failed update to evolve the microservices architecture to the target

Development faults

This section discusses the development faults of the DMU framework. The DMU framework implementation contains three parts: the framework core,

[8] According to the literature [74], the faults can be classified from eight criteria. In all the 256 combined fault classes, there are 31 likely combinations. All these combined faults belong to three overlapping groups: development faults, physical faults, and interaction faults.

the PaaS connectors, and the strategies implementation. The framework core is made of code that is stable, thus we consider that this code can be checked and verified. The PaaS connectors evolve with the PaaS API, which is also usually stable (i.e., backward-compatible). Thus, we believe that it is reasonable to assume a bug-free implementation of the framework core and the PaaS connectors.

As the part most probably to be customized by the DevOps team, the strategy implementation is not fully trusted. The following part of the section focuses on presenting how the DMU framework helps to handle the software faults in the strategy implementation.

During the development of a strategy, the strategy programmer can take advantage of the preview mode (Listing 4.8) provided by the DMU framework to test the correctness of the strategy. The preview mode can take mock inputs (i.e., initial and target architectures) to calculate the output (i.e., a sequence of intermediate architectures) produced by a strategy. Therefore, the strategy programmer can test the strategy by simply specifying the expected intermediate architectures for each typical case of initial and target architectures. The more inputs covered in the test, the fewer bugs hidden in the delivered strategy.

During the execution, the DMU framework handles an update request by iteratively calculating and applying the next intermediate architectures until reaching the target (Listing 4.7) as presented in Section 4.3.2. At each step, the strategy is in charge of producing the expected next architecture. A badly programmed strategy will either :

- raise an exception because it performs a bad action (e.g., null pointer exception) or lead to an interruption of the running update process (e.g., division by zero). In this case, the ongoing update is automatically stopped.

- produce an incorrect intermediate architecture, that cannot be applied by PaaS sites (e.g., the architecture contains two microservices with duplicate *id*). This invalid architecture is automatically detected by PaaS sites while applying the architecture.

- produce an endless sequence of intermediate architectures. This case may be automatically detected by the framework by processing the preview protocol (Listing 4.8) at the beginning of each update request. Moreover, an update request will be automatically stopped if it lasts more than a given delay, if such delay is given by the DevOps team [9]. In addition, the maximum number of intermediate architectures can also be given.

[9] The timeout value is configurable for each update request.

When a strategy is identified as having an erroneous behavior while processing a given update, the update is automatically stopped and, thanks to the kill-restart capability, the DevOps team is invited to restart the update with an adapted strategy and/or target architecture.

In summary, during the development, the preview mode of the DMU framework facilitates the debug of the strategy implementation. During the execution, the DMU framework also helps to detect the software flaws of strategies by catching the exceptions and pre-checking whether the specified target is attainable for the strategy. The framework follows a fail-fast principle [75] once detected any faults. That is, the framework stops the execution as soon as any unexpected error occurs. A stopped update can then be pursued through the kill-restart capability of the DMU framework.

Physical faults

This section discusses the physical faults during the update in terms of their types, their detection, and their repair. Physical faults encompass faults that can occur at any hardware involved in the update: the PaaS infrastructure, the DMU framework infrastructure, and the network connecting the framework to the PaaS sites. Based on the location of the faults with respect to the system boundary, such faults can be classified into two groups: external faults and internal faults (Figure 4.12). *External faults* originates outside of the framework (i.e., at PaaS sites or at the network connecting the framework to the PaaS sites). *Internal faults* arise inside the underlying infrastructure of the DMU framework.

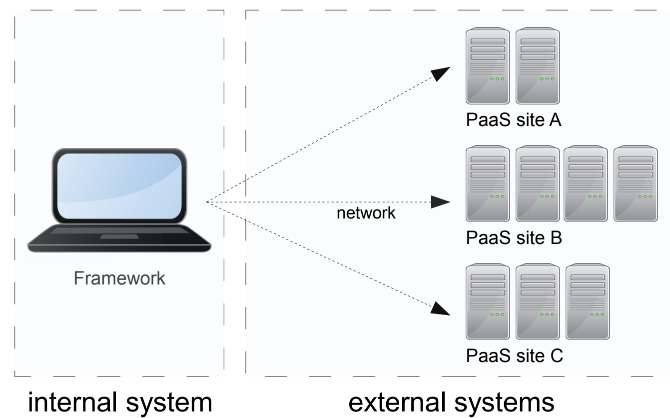


Figure 4.12: The classification of physical faults based on the location

To deal with physical faults, the DMU framework should firstly detect them. As previously said, PaaS sites are assumed to follow a fail-stop failure model [73], where any fault occurring at a PaaS site P leads to automatically stopping PS while ensuring the consistency of the micro-services P manages.

At the level of the framework, PaaS site faults are detected through using timeouts when issuing PaaS commands. Network faults are also detected through timeouts.

Regarding *internal faults*, we assume as well that the DMU framework follows a fail-stop failure model. Therefore, internal faults obviously cause the framework execution to be interrupted.

To recover from such faults when they occur at update time, the DevOps team needs to relaunch and continue the update process, relying again on the kill-restart capability of the DMU framework.

Interaction faults

Interaction faults are faults which occur at the elements interacting with the DMU framework during the update. As shown in Figure 4.13, an update process involves three actors: the DevOps team, the DMU framework, and the PaaS sites. The DevOps team issues the update command to the DMU framework. The DMU framework translates the update command into a sequence of PaaS operations. The PaaS sites apply the PaaS operations requested by the DMU framework. The interaction faults include the failures of the external systems (i.e., PaaS sites) and the mistakes of the user (i.e. the DevOps team).

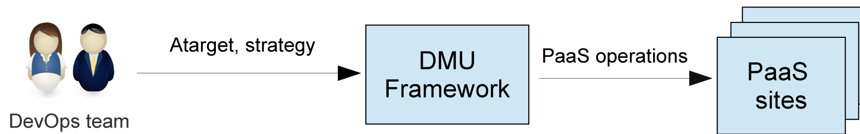


Figure 4.13: Involved software components in an update command

The failures of PaaS sites caused by hardware faults (i.e., external physical faults) have already been discussed in Section 4.5.2. The user mistakes are the operational mistakes conducted by the DevOps team during the update. These mistakes include the harmful human actions and the erroneous inputs.

- *Harmful human actions* are the DevOps team's actions which cause the update to be interrupted before reaching the target. These actions include starting or stopping the update command at the wrong time. In practice, it corresponds to the scenario of concurrently updating a microservice or accidentally stopping an ongoing update process.

  Notice that the detection of concurrent updates is not under the charge of the DMU framework. In practice, each microservice is managed by a specific DevOps team, which has its proper operator in charge of updating the microservice. Therefore, the organization and the operational disciplines of microservices can avoid the concurrent access. Regarding the harmful action of stopping an unfinished update, it does

not require the extra detection, as the framework execution is already stopped.

Overall, to repair the faults caused by harmful actions, a DevOps team simply needs to re-issue the same update command.

- The user inputs include the target microservices architecture and the strategy choice. Therefore, the *erroneous inputs* includes the following cases: an erroneous microservices architecture, flawed microservices codes or configurations, or a wrong strategy choice which does not deal with the specified update.

  Some inputs errors can be detected at the beginning of an update when processing the preview mode (e.g., an invalid target architecture). Others inputs errors will be detected when processing PaaS operations (e.g., a microservice unable to be started). Such errors leave the microservices into an intermediate architectural state that does not correspond to the target architecture. In any case, the DevOps team can exploit the kill-restart capability to pursue the interrupted update. Then, there are two possible choices here to pursue the update process. The first choice is to roll-back to the initial microservices architecture. In this case, the DevOps team simply specifies the initial architecture as target in the new update command. For such roll-back update, the strategy *Straight* is usually chosen. Otherwise, the DevOps team can change the strategy and/or roll-forward to a new target architecture.

4.5.3 Summary

The previous analysis involves three criteria of faults: the origin based on the system boundary (internal or external), the origin phase (development or operational), and the nature (hardware or software). Figure 4.14 shows the likely combinations of fault classes. Specifically, the origin of faults can be classified into three categories(Figure 4.15): the user (A), the DMU framework (B), and the external systems (C) [10].

All these faults eventually cause the interruption of the update [11], letting the updated application in an arbitrary architectural state, although consistent.

The DMU framework helps the DevOps team to repair these faults through its kill-restart capability. The faults at the external systems (#3, #4) and the hardware faults at the DMU framework (#2) can be repaired by re-issuing the same update command (presented in Section 4.5.2 and Section 4.5.2). Regarding the software faults at the DMU framework (#1), we concentrate on discussing the faults in the strategy implementation, which

[10] We separate the user from the external system, because the faults caused by the user-made mistakes usually requires specific repair methods (i.e. change user inputs).

[11] The faults detection is presented in the previous detailed analysis.

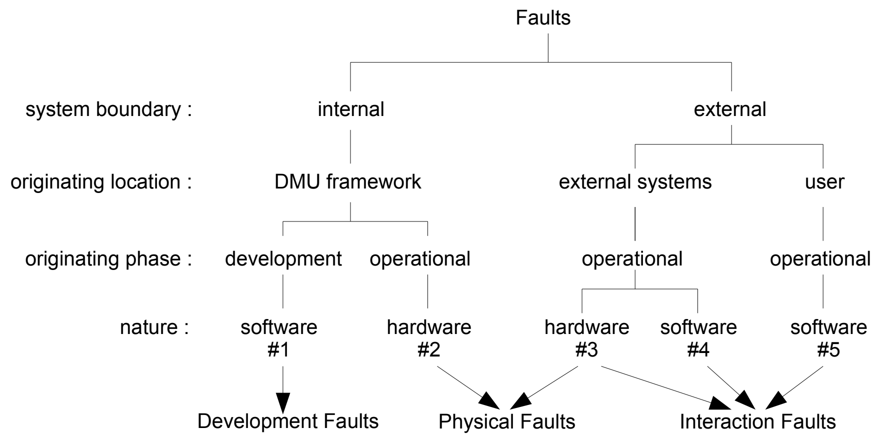


Figure 4.14: The classes of combined faults

can be repaired by re-issuing the update command with an alternate strategy (presented in Section 4.5.2). In terms of the faults caused by user mistakes (#5), depending on the fault reason (actions or inputs), it can be repaired by resuming, rolling-back, or rolling-forward the update (presented in Section 4.5.2). It is important to notice that all these repair processes can be applied by re-issuing another update command with the appropriate target architecture.

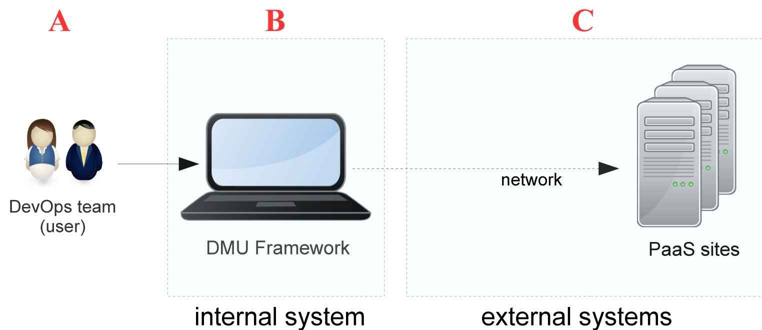


Figure 4.15: Origins of update faults

To sum up, whichever the type of faults, after it triggers the update to be interrupted, the DevOps team can exploit the kill-restart capability to repair the failed update. In the overall process of faults recovery, the faults still need human intervention to identify the fault origin, fix the error (e.g., correct the execution environment or the update command input), and decide the repair method. Once this is done, the DevOps team just has to relaunch the update process without questioning oneself. Due to the idempotency property, changes that have already been processed will not be processed again, and changes that have not yet been processed will now be processed, unless additional faults occur in which case the same

repair process will be applied again. This is how the DMU framework helps DevOps team to manage faults.

By comparison, when DevOps teams process manual updates and face a fault, they have to determine what changes have already been processed and compute what are the additional changes that need to be processed to reach the given target architecture. As the update of multiple microservices on multiple PaaS sites usually contains multiple operations executed in parallel, such manual repairs are exhausting and error-prone.

Chapter 5

Evaluation

Contents

5.1 SLA protection **65**
5.1.1 Lizard application 65
5.1.2 Account application 68
5.2 Robustness **70**
5.2.1 Network faults 72
5.2.2 Update process faults 74
5.2.3 Erroneous strategy 75
5.2.4 Microservice faults 78
5.3 Ease of use **79**
5.3.1 Programming Strategies 80
5.3.2 Updating Microservices 81
5.3.3 Comparison with an imperative approach 83

This chapter evaluates the proposed DMU framework from three aspects: the SLA protection, the robustness, and the ease of use.

5.1 SLA protection

This section reports on how the DMU framework protects SLA constraints through two use-cases: a production microservices application *Lizard* and an open-source microservices application *Account*.

5.1.1 Lizard application

Lizard is a cross-canal order capture application composed of four microservices (*Shop Front*, *Catalog*, *Eligibility* and *Basket*). Figure 5.1 shows the architecture of *Lizard*. The application is about 10000 lines programmed in Java / Angular.

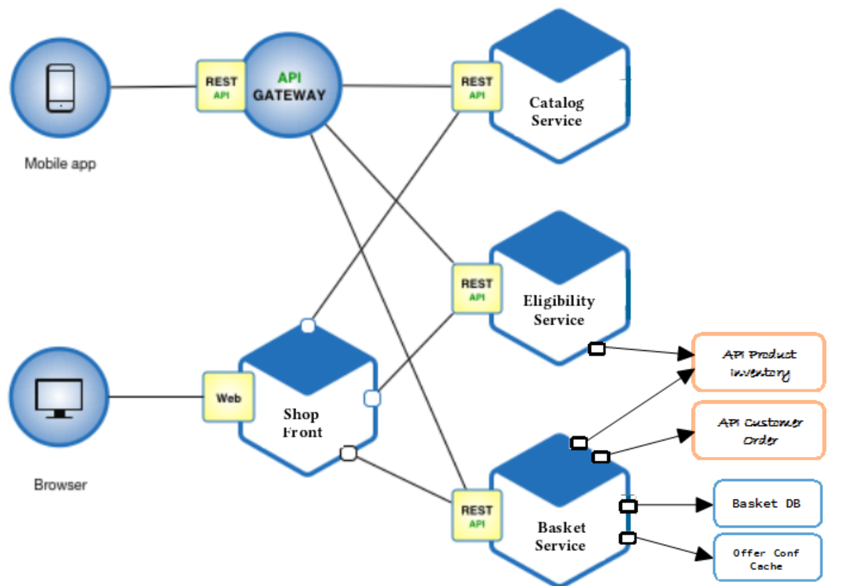


Figure 5.1: The architecture of the use case *Lizard*

We use the DMU framework to update a complete clone of *Lizard* in live production at Orange, focusing on the simultaneous update of two microservices (*(C)atalog* and *(E)ligibility*) that are deployed redundantly over three distributed CloudFoundry PaaS sites (S_a, S_b and S_c). The update to perform includes a code upgrade for the *Catalog* microservice and configuration changes for the *Eligibility* microservice.

Cloud Foundry is used in version 2.75.0, running on the Cloudwatt cloud [76] on top of the OpenStack IaaS [31], under VMs with medium flavor (2vCPUs, 4GB RAM, 50GB disk). The update framework is launched from a remote computer (Intel i7-7820HQ @ 2.90 GHz, 16GB RAM, 500GB SSD disk).

Four strategies are experimented to process such update, comparing their behavior and impacts (in terms of time efficiency, financial cost, and mi-

croservices availability). The strategies impacts are measured with the following metrics:

- time efficiency: the duration of the whole update process
- financial cost: the additional resource consumption (representative of the billing costs [1]) for the update.
- microservices availability: the rate of rejected microservices requests during the update. During the update, we monitor the microservices performance under their anticipated peak load condition (10000 requests per minute) with *Apache Jmeter*.

Each experiment is repeated 30 times to get the average data. Results are given in Figure 5.2.

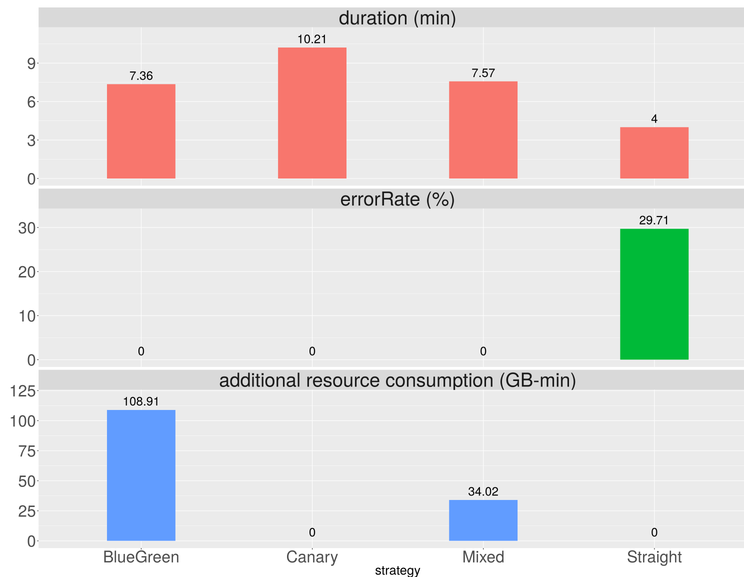


Figure 5.2: Update metrics of the *Lizard* application use-case

The *Canary* strategy (whose behavior is shown in Figure 5.3) does not involve an additional cost in terms of resource usage. It ensures that any deployment of a 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 with a temporary route to allow testing before continuing the update. Notice that although not visible in the figure, some client requests are not correctly served (about 0.007%), as the experiments are performed under the peak load condition and the microservices have one less instance running during the update. Overall, this strategy involves 21 intermediate

[1]In the experiments, the resource cost is calculated by multiplying the number of instances by their memory size (in GB) by total duration (in minutes), as it is the billing model used by most Cloud Foundry services.

architectures to reach the target, 7 per site, sites being updated sequentially. Its processing took about 10 minutes.

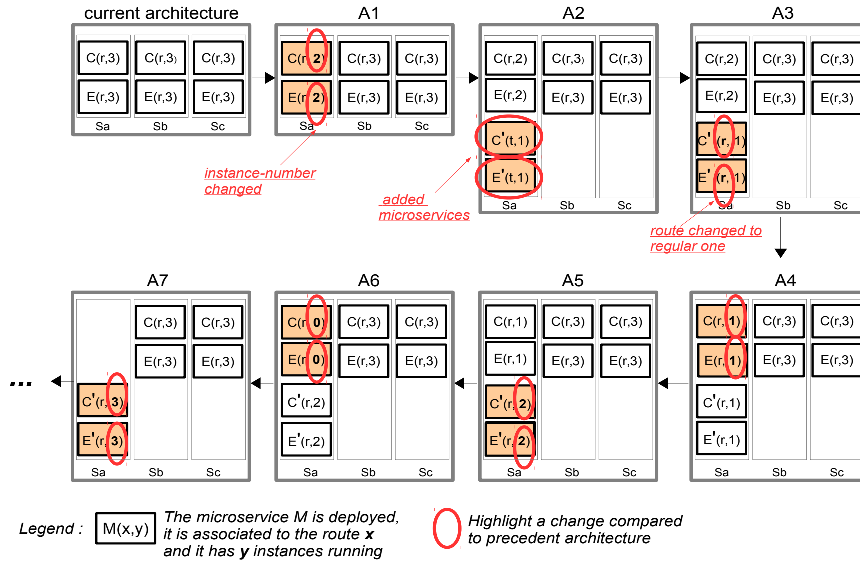


Figure 5.3: Real application update through the Canary strategy

In comparison, the *BlueGreen* strategy (Figure 5.4) 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 the same availability but consumes nearly the double resource during the update. The duration is about 7 minutes, 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. In the practice, *BlueGreen* strategy also brings another benefit: enabling the DevOps team to perform load tests on the new version when arriving the first intermediate architecture A_1.

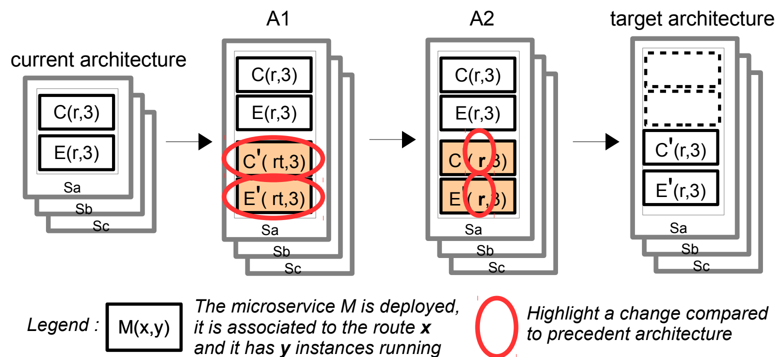


Figure 5.4: Real application update through the BlueGreen strategy

The *Mixed* strategy (Figure 5.5) updates microservices instance by in-

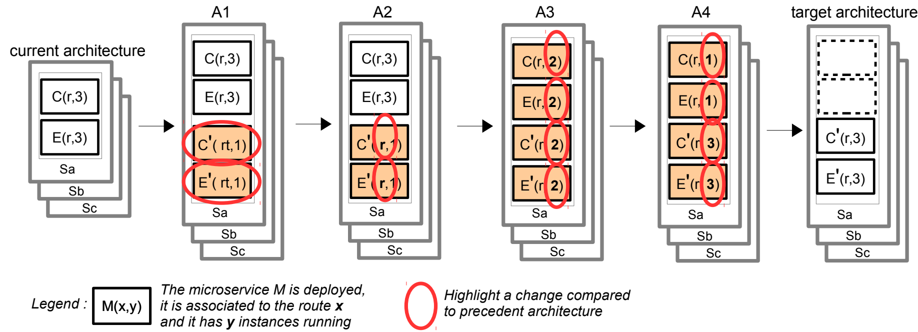


Figure 5.5: Real application update through the Mixed strategy

stance, 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. However, the DevOps team cannot load test the new version microservice with the *Mixed* strategy.

Finally, the *Straight* strategy delivers the shortest update duration (4 minutes), 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 (29.71% requests failed) as the microservices are stopped before their new version is uploaded, recompiled, and restarted.

5.1.2 Account application

Another use case is an open-source microservices-based application named *Account* [77]. As shown in Figure 5.6, the application consists of two microservices: *web* and *account*. The microservice *web* is deployed with 120 instances, and *account* with 150 instances over three Cloud Foundry sites. We experiment through updating the code of the microservice *account* with various strategies.

During the experiments, we find that the *Straight* strategy and the *BlueGreen* strategy cannot properly update the microservice *account* in our experiment environment. The update either takes several hours or never succeeds to start all the instances. The reason is that our Cloud Foundry platforms limit the number of concurrently starting instances, as they have limited resource. According to Cloud Foundry documentation [79], if too many instances are starting concurrently, they are slow to start or do not start successfully at all.

Both the *Straight* strategy and the *BlueGreen* strategy contain a step

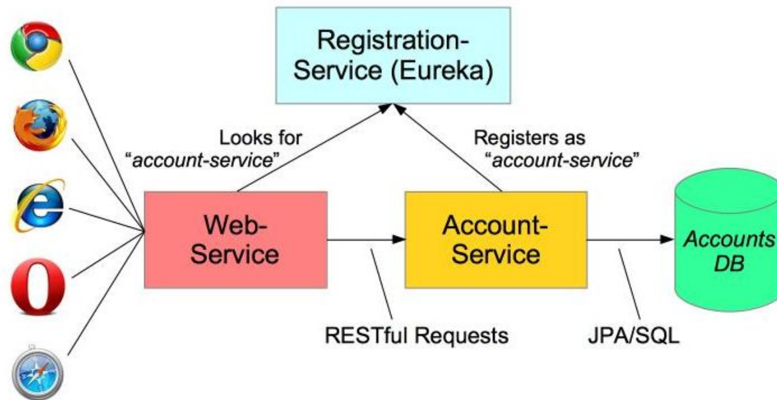


Figure 5.6: The architecture of the use case *Account*. Reprinted from [78]

that concurrently start 150 instances on each site, which causes the Cloud Foundry platforms failing to start all the instances within a reasonable duration. Therefore, this application needs to be updated with the strategy *Canary* or *Mix*, that incrementally starts the new instances. We configure these strategies to incrementally update 20 instances each time.

Figure 5.7 shows the experiments result for the update with the strategy *Canary* and *Mixed*. As shown, these two incremental update strategy can correctly perform the update within 30 minutes. Similar to the experiments with the use case *Lizard*, the strategy *Canary* consumes more resource comparing to the strategy *Mixed* at the cost of slightly reduced availability.

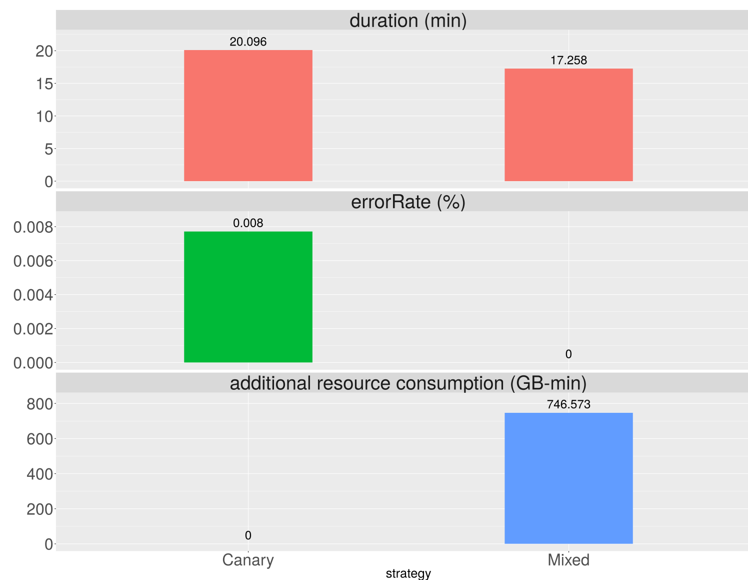


Figure 5.7: Update metrics of the use case *Account* with two strategies

5.2 Robustness

This section evaluates our DMU framework under faults. For each type of faults that may be encountered, we experiment a typical fault scenario.

The faults can originate from three locations (shown in Figure 4.15): the user, the DMU framework, or the external systems. The (end) user of the DMU framework is the DevOps team that issues update requests to manage microservices. The DMU framework includes both the framework software and its underlying infrastructure. The external systems include the PaaS sites involved in the update and the network connecting the DMU framework to the PaaS sites.

Table 5.1 presents the different faults, the way we triggered them in our experiments, and the expected handling behaviors for each type of faults. Typically, the *external system faults* are exemplified by the outage of the network between the DMU framework and the PaaS sites. The general handling behavior is, for the DMU framework, to detect the fault, stop the update execution (i.e., follow the fail-stop failure model), and then accept new update requests to repair the fault.

The *internal faults* (i.e., faults at the DMU framework) are divided into hardware faults and software faults, as they require different recovery processes. As presented in Section 4.5.2, the internal hardware faults naturally stop the execution of the DMU framework. They are expected to be recovered by resuming the interrupted update, through re-issuing the same update request. In the following experiments, the internal hardware faults are triggered by killing the process of the DMU framework during the update.

The *internal software faults* refer to the development faults of the DMU framework. As presented in Section 4.5.2, the framework core and the PaaS connectors can be considered as trusted code in the DMU framework software, while the strategies (except the predefined ones) cannot. Therefore, the internal software faults are experimented with an erroneous strategy. To handle internal software faults, the DMU framework first needs to detect the faults and stop the execution. Then the DevOps team is supposed to be able to roll-forward the update by issuing an update request with an adapted strategy.

As presented in Section 4.5.2, the *user faults* can be categorized into two types: erroneous user inputs and harmful human actions. The input of the DMU framework update request contains the target microservices architecture and the chosen strategy. The *erroneous user inputs* are shown with a common scenario: an incorrect microservice configuration, which leads to an inability to start it. Two recovery methods (roll-back and roll-forward) for this fault are both experimented. The *harmful human actions* are experimented with the killing of an update process, which is expected to be fixed by resuming the interrupted update.

Table 5.1: experimented faults scenario

Fault class	Example scenario	Framework Expectation
external system faults (#3, #4)	network outage	fail-stop + kill-restart (resume)
internal hardware faults (#2)	kill the update process	kill-restart (resume)
internal software faults (#1)	badly programmed strategy	fail-stop + kill-restart (roll-forward)
erroneous user inputs (#5)	erroneous microservice code	fail-stop + kill-restart (roll-back or roll-forward)
harmful human actions (#5)	kill the update process	kill-restart (resume)

In the following experiments of faults, we perform the same update request on the production application *Lizard* as the SLA experiments in Section 5.1. The faults experiments use the strategy *Mixed* (Figure 5.5). The *Mixed* strategy performs a complex update workflow to satisfy the requirement on both availability and resource consumption. It can be seen as a typical user-defined strategy.

Similarly to the SLA experiments environment, the microservices are deployed on three Cloud Foundry sites in the following experiments. The DMU framework is executed on a VM (2 vCPUs, 7.8GB RAM, 50GB SSD Disk) on the Cloudwatt cloud [76] with the monitoring processes and the faults simulation processes.

To evaluate the behavior of the DMU framework in case of failures, we firstly verify whether the microservices deployment architecture stays reconfigurable through our framework, which allows to reach the desired target at the end (assuming this architecture is not an erroneous one).

In addition, we will check whether the update process satisfies its SLA constraints in our experiments, but it is important to recall that the SLA constraints of an application may be unsatisfied when facing failures at update time. To check SLA constraints, we monitor the microservice performances and its resource consumption during the whole update [2]. In each fault experiment, the average update duration is measured and compared to the case without faults. As the experiments run in the real production environment, the execution time of each PaaS operation is uncontrollable. Therefore each experiment is performed 10 times to get the average execution time of the update process.

The following sections evaluate the DMU framework in the four typical fault scenarios. Section 5.2.1 experiments the update under the network faults. Section 5.2.2 kills the framework process during the update. Section 5.2.3 performs the update with an example erroneous strategy. Sec-

[2]In the experiments, the intended update request (update the microservices application *Lizard* with the strategy *Mixed*) requires the following SLA: no downtime and less than 24 GB memory consumption.

tion 5.2.4 experiments the update with erroneous microservice code.

5.2.1 Network faults

This section evaluates how the DMU framework resists network faults. As presented in Section 4.5.2, the network faults are supposed to interrupt the framework execution. After the failure of this update request, the DMU framework is supposed to be able to resume the update by re-processing the same update request.

We have experimented the network faults in two modes : a single fault or recurrent faults. The single fault only appear once during an experiment. The recurrent faults keep occurring until the experiment finishes. Besides the occurring mode (single or recurrent), the behavior of the fault simulation process also depends on two parameters: the fault-free duration and the fault duration (Figure 5.8). The fault simulation process waits for the *fault-free duration* before cutting the network. Then, it waits for the *fault duration* before reconnecting the network. Note that, the parameters (fault-free duration and fault duration) values remain unchanged during an experiment of recurrent faults.

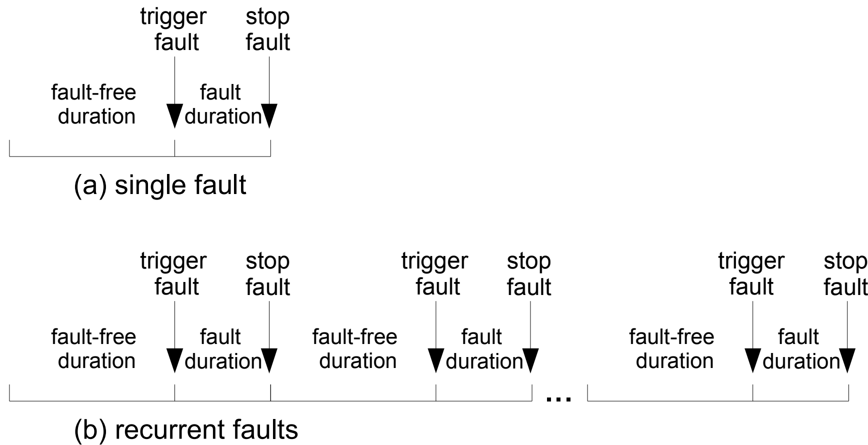


Figure 5.8: The parameters of the network faults experiments

In the experiment implementation, the faults simulation process is started once the update request is issued. To simulate the network faults, the command *iptables* is used to cut the network connection of the DMU framework. The *fault duration* takes three values (1s, 15s, 60s). To trigger the fault at an arbitrary moment during the update, the *fault-free duration* takes a random value between one second and six minutes, because 95% of the update process takes less than six minutes in the case of no fault [3]. Once we detect that the update process is stopped with failures, we wait

[3] In the data of final experiments, we remove the cases in which the fault occurs after the update finishes.

one minute (ensuring the network recovered) then re-issue the same update request. Note that, in case of recurrent faults, the re-issued update can also encounter faults. An experiment is marked as failed if unable to deliver the target architecture within ten times of restarting the update.

The experiment results are presented in Table 5.2. The table summarizes the key metrics compared to the case without fault. In this case, the microservices provide nearly no downtime and consume a maximum of 24 GB resources during the update. This means that the *Mixed* strategy correctly satisfies the SLA requirements even in the case of network faults.

All the experiments with single faults reach at the target architecture. However, the experiments with recurrent faults fail to reach the target architecture in the case where the fault-free duration is less than one minute. That is, the update cannot be completed in the case of too frequent faults. After analyzing the failing messages, the failed updates block at the microservice code upload operation. Because the upload operation takes 67 seconds on average, this operation can not finish in these cases. It blocks the advance of the update. In such an environment where fault-free duration is always shorter than an operation execution time, it is impossible to finish the update. These faults cases require the DevOps team to fix the hardware execution environment.

As shown in Table 5.2, the overall update process takes 6 minutes 19 seconds on average in the case of the single fault and 6 minutes 51 seconds in the case of recurrent faults. Compare to the case without fault (5 minutes 50 seconds), it takes approximately less than one minute to repair the fault.

Table 5.2: experiment result of updating with network faults

	no fault	single net fault	recurrent net fault
average percentage of rejected microservices requests	0.009%	0.017 %	0.024 %
maximum microservices memory consumption	24 GB	24 GB	24 GB
percent of experiments correctly reach A_{target}	100%	100%	93.3 %
average execution duration	5M50S	6M19S	6M51S

Figure 5.9 and Figure 5.10 show the update execution time when the fault occurs at different moments and takes different durations. In the case of the single fault, both the fault occurring moment and the fault duration do not impact much on the execution time. In the case of recurrent faults, more frequent the faults occur (shown as smaller fault-free duration in Figure 5.10), more time the update takes.

To conclude, the DMU framework can resist the network faults when the execution environment still leaves enough fault-free duration for each PaaS operation. Through the kill-restart capability, the DMU framework can correctly repair the failed update caused by the network faults.

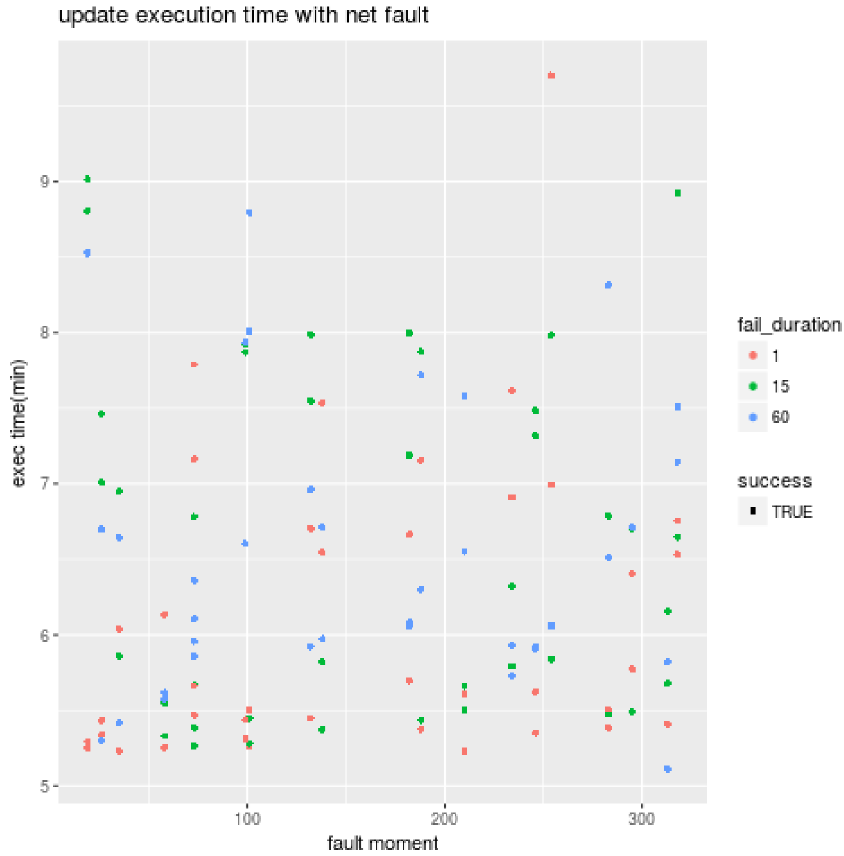


Figure 5.9: Update execution time in the case of single network fault

5.2.2 Update process faults

This section considers both internal hardware faults (e.g., the server running the DMU framework has a power outage during the update) and harmful human action faults (e.g., the user accidentally stops the DMU framework during the update). To experiment with such faults, we add a process in charge of killing an update process at a random moment. The faults trigger pattern is similar to the network faults (shown in Figure 5.8), except that the *fault duration* is considered to be negligible. Once an update is interrupted, the DevOps team is supposed to be able to resume the update through re-issuing the same update request.

Table 5.3 shows the experiment result. The update process works correctly (reached the target at the end, no downtime, limited resource consumption) in the case of only killing the framework process once. Similar to the network faults, the experiment fails to reach the target architecture in the case of frequent recurrent faults.

Figure 5.11 and Figure 5.12 show the update execution time when the

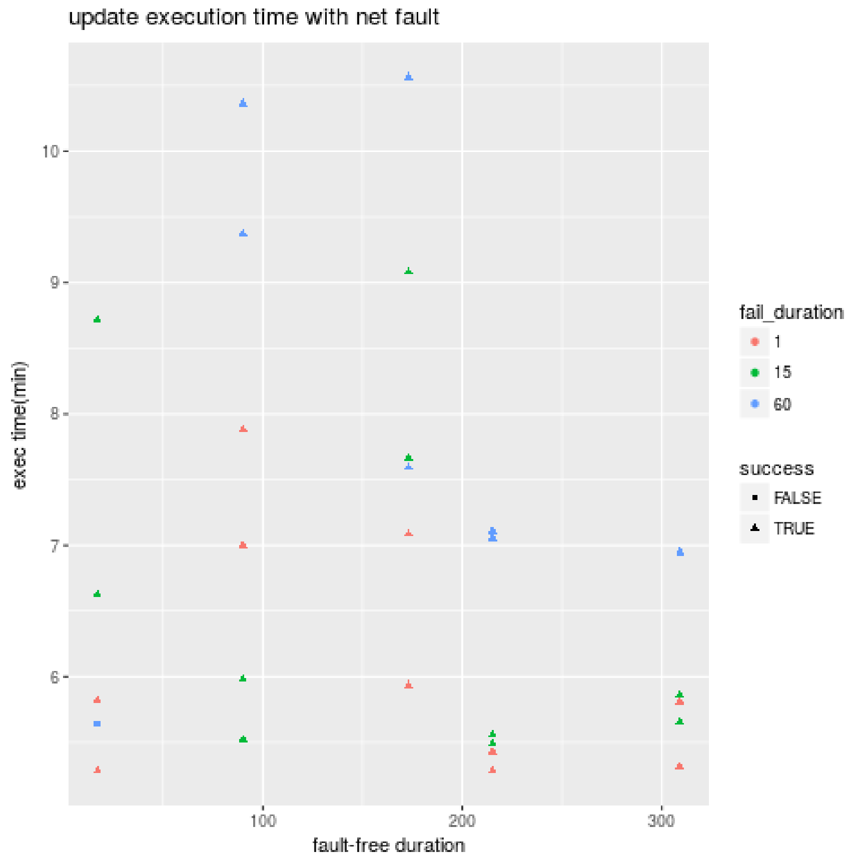


Figure 5.10: Update execution time in the case of recurrent network faults

fault occurs at different moments and takes different durations. In the case of the single fault, the fault occurring at the first half of the update may take longer execution time. The reason is that the longest operation (microservice code upload) executes in the first half of the update. Killing the update process during the upload operation causes it to be stopped, and requires it to be restarted from the beginning in the following update request. In the case of recurrent faults, the update failed to reach the target architecture in the case of the frequent faults, as the same reason presented in Section 5.2.1.

To conclude, the DMU framework supports the fail-stop process faults, as long as the intervals between the faults leave enough time for progressing in the update.

5.2.3 Erroneous strategy

The section experiments with two erroneous strategies:

- $MixedError1$ that returns an invalid next architecture

Table 5.3: experiment result of killing the update process during the update

	no fault	single fault	recurrent faults
average percentage of rejected microservices requests	0.009%	0.012%	0.021%
maximum microservices memory consumption	24 GB	24 GB	24 GB
percent of experiments correctly reach A_{target}	100%	100%	70 %
average execution duration	5M50S	6M11S	6M23S

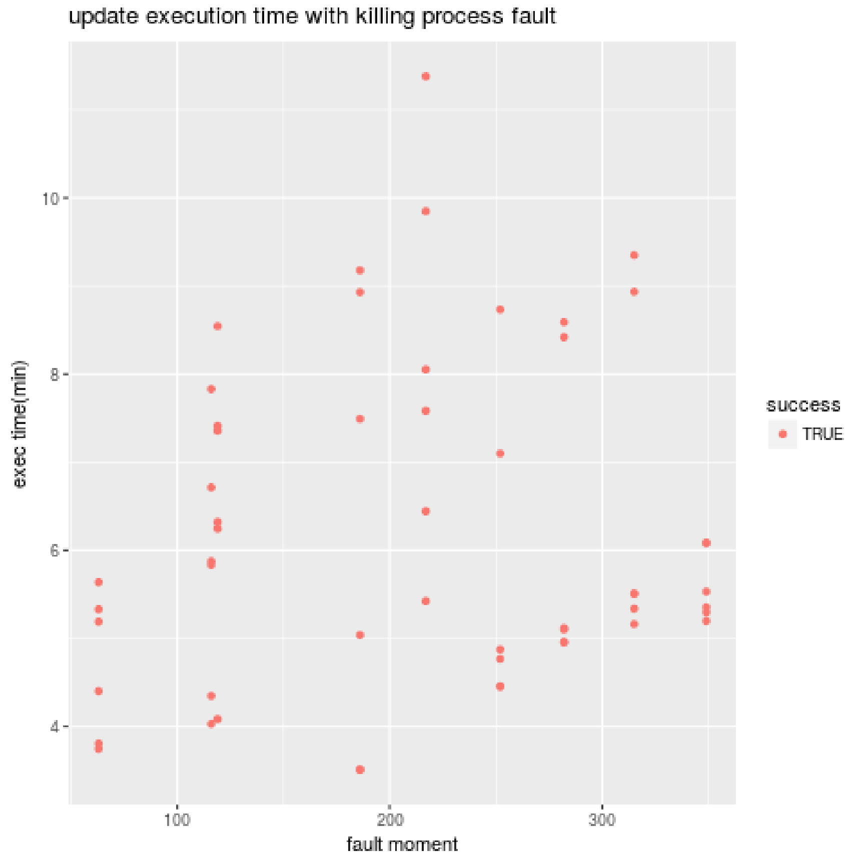


Figure 5.11: Update execution time in the case of single killing process fault

- $MixedError2$ that outputs intermediate architectures that never reach the target

As presented in Section 4.5.2, the DMU framework should detect the fault and interrupt the update process. After update interruption, the DMU framework is supposed to allow the DevOps team to reissue an update request with another strategy and/or another target architecture.

In the first experiment, we use an erroneous strategy $MixedError1$ that accidentally specifies the same *id* for two versions of a microservice. At

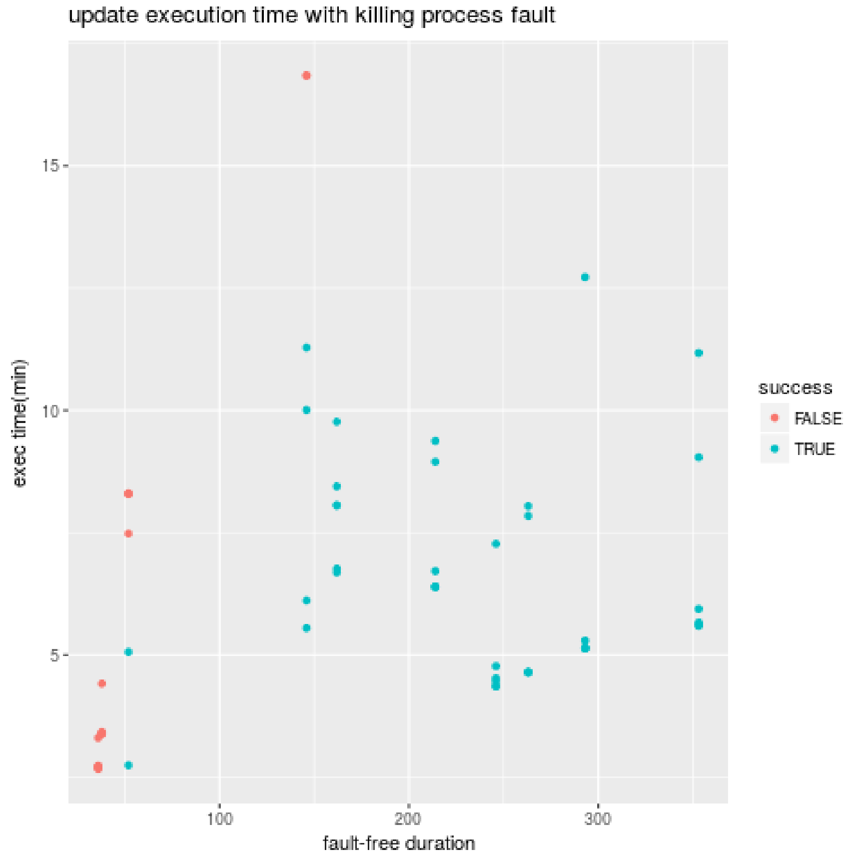


Figure 5.12: Update execution time in the case of recurrent killing process faults

update time, this fault is detected by the PaaS platform. Then, another update request is re-issued by the DevOps team with a correct strategy (the $Mixed$ one), which allows to reach the target architecture.

The erroneous strategy $MixedError2$ returns an endless sequence of intermediate architectures as shown in Figure 5.13 that causes the update process to fall into an endless loop. The framework simply detects the error while processing the preview protocol (unable to reach the target within the limited number of iterations) at the beginning of the update request.

Table 5.4 presents the experiments results. As shown, all the experiments finally reach the target architecture. Compared to the experiments without faults, the additional execution time of the experiments with $MixedError1$ (11s on average) and $MixedError2$ (13s on average) is spent on executing the erroneous update request, which includes retrieving the current architecture and detecting the error. In these two specific faults, as the faults are detected before applying PaaS operations, the erroneous strategy does not

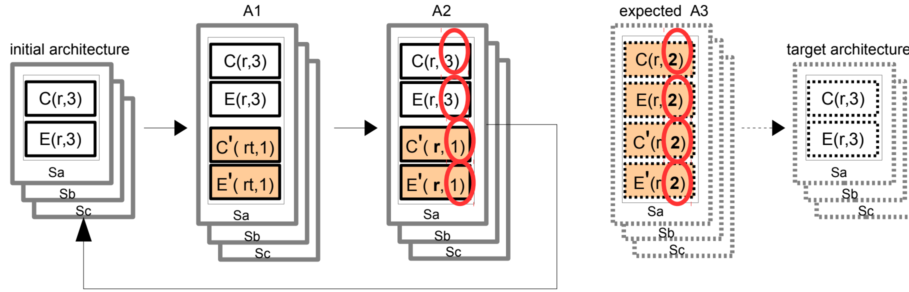


Figure 5.13: Update through the erroneous strategy $MixedError2$ (endless loop)

impact the microservices deployment architecture and its SLA properties.

Table 5.4: experiment result: update with an erroneous strategy

	no fault	invalid A_{next}	endless loop
percent of experiments correctly reach A_{target}	100%	100%	100%
average percentage of rejected microservices requests	0.009%	0.012%	0.005%
maximum microservices memory consumption	24 GB	24 GB	24 GB
average execution duration	5M50S	6M01S	6M03S

5.2.4 Microservice faults

We consider here erroneous microservices configurations in the target architecture, making microservices unable to start. The DevOps team has two options to repair such a fault: *roll-back* to the initial architecture or *roll-forward* to another target architecture. Section 5.2.4 demonstrates the roll-back remedy, and Section 5.2.4 illustrates the roll-forward remedy.

Roll-back the update

This section evaluates how the framework allows to roll-back the update in the case of an erroneous target architecture. In the experiment, the DevOps team first issues an update request with an erroneous microservice configuration for the microservice *Eligibility* [4]. The erroneous configuration causes the microservice to fail when starting on both three sites, interrupting the update process.

To roll-back to the initial architecture, the DevOps team just re-issue another update request which specifies the initial architecture as the target, and chooses the *Straight* strategy as the roll-back workflow.

[4]As presented in Section 5.1, the experimented microservices application contains two microservices: *Catalog* and *Eligibility*

As shown in Table 5.5, this request correctly evolves the microservices architecture to its initial state. The overall update-rollback process takes 4 minutes 29 seconds on average. During the process, the remedy process (roll-back) takes 19.23 seconds on average.

Table 5.5: Result of the experiment: roll-back from the microservice error

	update-rollback
final architecture	$A_{initial}$
average percentage of rejected microservices requests	0.0001%
maximum microservices memory consumption	24 GB
total execution duration (average)	4M29S
average remedy duration	19.23S

Roll-forward the update

Similarly to the roll-back experiment, this section reports on using a rollforward approach in case of an unreachable architecture.

The initial erroneous update request and its detection is the same as with the roll-back experiment. Upon the interruption of the update process, the DevOps team re-issue another update request with a new target architecture, and still chooses the *Mixed* strategy.

As shown in Table 5.6, the update process correctly reaches this new target. The overall update-rollforward process takes 5 minutes 47 seconds on average. During the process, the remedy process (roll-forward) takes 1 minutes 53 seconds on average. Comparing to the case of roll-back then restart which takes 6 minutes 9 seconds, the roll-forward remedy is about 3 times faster.

Table 5.6: Roll-forward from the microservice error

	update-rollback
average percentage of rejected microservices requests	0.026%
maximum microservices memory consumption	24 GB
total execution duration (average)	5M47S
average remedy duration	1M53S

5.3 Ease of use

As presented in Section 4.1, the update management involves three roles (microservices manager, strategy programmer and the PaaS connector programmer). Usually, the DevOps team only plays the role of microservices manager which issues update requests daily. In case of specific SLA requirements, the DevOps team may also play the role of strategy programmer.

In this section, we report on using the DMU framework, mainly evaluating (1) the ease of programming strategies (Section 5.3.1) and (2) the ease of updating distributed microservices applications (Section 5.3.2). Section 5.3.3 compares the DMU framework with a script-based approach.

5.3.1 Programming Strategies

The DMU framework is adaptable to any strategy. This section evaluates the required efforts for programming strategies with the DMU framework.

The strategy programmer can define his/her proper strategies through implementing the strategy interface (Listing 4.9), as demonstrated in Section 4.4.1. Besides programming a strategy from scratch, the strategy programmer can also customize the existing strategies.

In particular, one possibility is to play with the order of transitions. For instance, the update case shown in Figure 4.6a can be easily implemented based on the *AddRemove* strategy (Listing 4.10). By just inverting the transitions ordering (i.e., *transitions* is then set to the sequence $newList(Tremove, Tadd)$), the strategy programmer implements a new strategy *RemoveAdd*. In the previously considered scenario of migration, the *RemoveAdd* strategy generates a different path made up with two intermediate architectures: the first one aims at removing m_1 and m_2 on $site_1$ while the second one purposes to add m_1 and m_2 on $site_2$.

Moreover, new strategies can also be defined through changing the transitions they use. For example, we defined a *BlueGreenByGrp* variant of the *BlueGreen* strategy. The *BlueGreenByGrp* strategy splits the microservices to update on each site into groups, updating groups incrementally in order to limit the number of microservices being simultaneously deployed per site. To program this new strategy, we only had to implement an alternate $Tupdate$ transition, updating at most k microservices per site at each step. The other transitions ($Tadd$, $Tswitch$, $Tremove$) can be directly reused.

In addition to customizing the transitions order, the DevOps team can also define the order of updating sites. For instance, through configuring the order as $[[site_1], [site_2, site_3], [site_4, site_5]]$, the framework waits the update finished at $site_1$, then updates $site_2$ and $site_3$ in parallel, finally updates $site_4$ and $site_5$ in parallel. The implementation of customizing sites order is based on the sub-architecture. A sub-architecture is a part of the microservices architecture containing some site(s). The DMU framework performs the update fix-point (i.e., processing the transitions) on each sub-architecture. Therefore, the DevOps team can easily customize a strategy which updates the site in the specified order.

Additionally to the presented strategy (e.g., *BlueGreen*), we programmed a dozen of other strategies which are the most popular ones in the industry, summarized in Table 5.7. These strategies are composed of a few transitions (from 1 to 6, see the column named *Tr.*). Altogether,

they require programming about fifteen transitions (some listed in Table 5.8 below).

Overall, programming transitions is easy and short in terms of the number of lines: (i) all transitions are programmed the same way, comparing the current and target architectures to determine the next architecture to reach, (ii) all transitions are programmed in about 5 to 20 lines of code.

Table 5.7: Some strategies programmed

Name	**Description**	**No. Tr.**
Straight	reach target directly (no intermediate architecture)	1
Deploy	deploy all in target, one microservice at a time	1
DeployBySite	deploy all in target, site by site, all microservices in parallel on a site	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 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) before applying Canary strategy for pending instances	5

Table 5.8: Some transitions programmed

Name	**Description**	**LOC**
Tremove	undeploys microservices removed in target	10
Tadd	deploys microservices added in target	13
Tupdate	deploy green versions for microservices updated in target	15
Tupdate-inc	as Tupdate, k microservices at a time	16
Tupdate-ip	update microservices *in place* (stopping then restarting them)	8
Tswitch	switch green microservices from temporary routes to regular ones	16
Tscale	scale microservices	11
Tclean	remove all microservices in current	3
Tdeploy	deploy all microservices in target	6

5.3.2 Updating Microservices

This section evaluates the required management effort of updating a microservices application with the DMU framework during the lifecycle of a microservices application. Taken the previous example described in Figure 4.8, we report here three update cases: the initial deployment (Figure 5.14a), the update (Figure 5.14b), and the step-by-step update (Figure 5.14c).

At first, the DevOps team of this microservices application A needs to deploy the application as A_{init} described in Figure 5.14a. To perform the

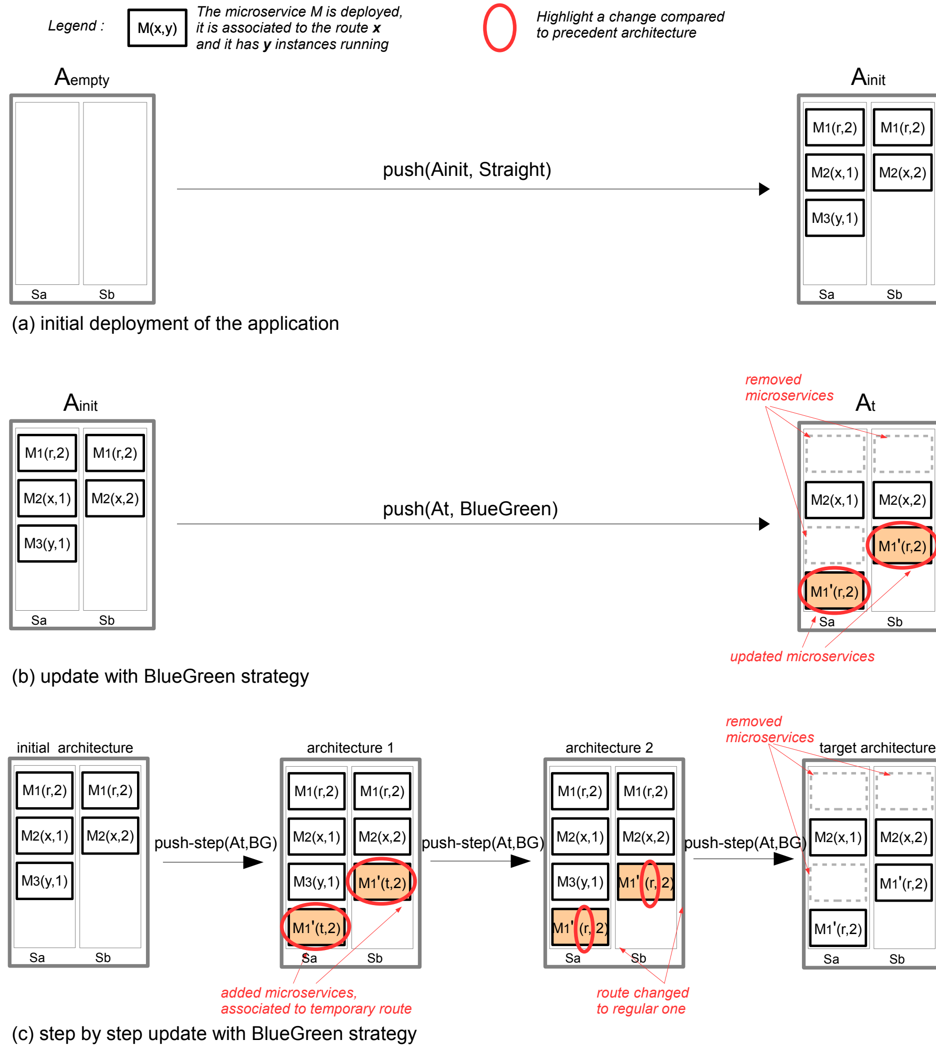


Figure 5.14: Updating with the BlueGreen strategy

initial deployment of this application, we simply use the *push* command with the *Straight* strategy, giving as target the architecture A_{init}. This initial architecture A_{init} specifies that A is composed of two microservices M_1 and M_2 to deploy on two (Cloud Foundry, version 2.75.0) PaaS sites Sa and Sb. Additionally, a microservice M_3 should be deployed on Sa. After describing the *initial architecture*, the DevOps team simply needs to launch the following command:$push(A_{init}, Straight)$.

Then, the DevOps team decides to use the *BlueGreen* strategy for updating the application A to the target architecture (A_t) shown in Figure 5.14b. Similarly, the DevOps team simply defines the target architecture A_t and makes the following request: $push(A_t, BlueGreen)$.

In addition, the DevOps team may also play the update in the ”step by step” mode (Figure 5.14c). The ”step by step” mode pauses the update after delivering each intermediate architecture. This mode allows the DevOps team to manually control update progress in each step. For example, the DevOps team may wish to process the last transition ($Tremove$) only after they have checked that DNS changes have been propagated, meaning that the *urls* used to access the old microservices will no more be used. The ”step by step” mode fulfills this checking need. After having done the necessary checks, the DevOps team can easily resume the update execution by re-issuing the same *push* command.

5.3.3 Comparison with an imperative approach

This section compares the proposed DMU framework with an imperative automation approach. We compare the two approaches with regard to the ease of programming a strategy and the ease of updating a microservices application.

The complexity of the update automation depends on the supported update cases. Therefore, our evaluation starts with a specific use case, then discusses more general cases.

In the following of this section, we automate the strategy $Mixed$ with two approaches. The strategy $Mixed$ (presented in Table 5.7) first creates one instance of the new version, then gradually replace the old version instances with the new version, all sites being updated in parallel.

We start from the simplest case: updating the code (a specific change type) of two microservices (a specific microservices application) deployed on two Cloud Foundry sites (a specific PaaS solution).

The easiest implementation of the imperative approach is to write a script with some knowledge of the microservice application. That is, microservices architecture (which microservices are deployed on which sites) and some microservices properties (e.g., microservice name, route) are hard coded in the script. In this case, the script needs to construct an update workflow with the Cloud Foundry operations [5]. Currently, the imperative approach has nearly 40 lines of code.

Then, we want to update the microservices across various PaaS solutions. For example, the use case now involves two sites: Cloud Foundry and Kubernetes. The script needs to identify the PaaS solution for each site and to call the corresponding operations for the five steps of the update workflow. In this case, the script needs nearly 70 lines of code.

Next, we want to make the script reusable across various microservices applications. Therefore, the script needs to separate all the application-dependent parameters. As it involves many parameters (e.g., name, route

[5]This update process involves five steps (seven Cloud Foundry operations) for each microservice on each site.

and instance number of each microservice), the script needs to parse the description file of the microservices architecture, so that it can be easily used in the daily update. It takes nearly 110 lines of code.

Finally, we want the update script to support various change types (e.g., architectural changes, code changes, configuration changes). The script needs to retrieve the current architecture, compare the difference with the target architecture, and decide the PaaS operations for each change type. The script does not necessarily need to automate all the changes cases. It can automate only the common cases and return with error messages for untreated cases. These untreated cases are left to be handled during the daily update. Depending on the choice of the DevOps team, the script needs about 140 to 210 additional lines of code.

With the DMU framework, the strategy programmer first needs to identify the needed transitions ($Tadd$, $Tupdate$, $Tswitch$, $Tscale$, and $Tremove$). As the strategy $Mixed$ behaves similarly as the existing strategy $Canary$, the strategy programmer can simply extend the strategy $Canary$ to reuse four of its transitions: $Tadd$, $Tupdate$, $Tswitch$, and $Tremove$. Then the strategy programmer only needs to implement the transition $Tscale$, which is in charge of scaling down the old version instances and scaling up the new version. The strategy $Mixed$ implementation takes only 28 lines of code.

Strategies defined with the DMU framework are PaaS-independent and architecture-independent. That is, they are reusable across various PaaS solutions, various microservices applications, and various change types.

Table 5.9 summarizes the comparison of the two methods. Regarding the programming effort, the script-based approach needs to produce longer code compared to the DMU framework, especially when assuming a same level of functionality. The additional complexity is strongly due to the fact that the script programmer has to manage by itself the heterogeneity of PaaS operations in its script. Especially, the script programmer needs to find the corresponding PaaS operations for each step, then compose them into parallel or serial workflows according to the previous specification.

Table 5.9: Comparison of the ease-of-use between the DMU framework and the script

	DMU framework	script (specific)	script (indep. PaaS)	script (indep. app)	script (indep. change)
required knowledge	strategy workflow; strategy prog. model;	strategy workflow; script prog.; CF operations; multi-process prog.	strategy workflow; script prog.; CF operations; multi-process prog.; K8s operations;	strategy workflow; script prog.; CF operations; multi-process prog.; K8s operations; parse description file;	strategy workflow; script prog.; CF operations; multi-process prog.; K8s operations; complex change cases managing;
lines of code	28	40	70	110	140 - 210

The framework implementation contains three parts: the framework core, the PaaS connector, and the strategy implementation. Table 5.10 shows the lines of code of each part. Supporting an additional PaaS solu-

tion only requires to implement a PaaS connector. Customizing a strategy only requires to code a strategy. The framework core is always reusable for any update case.

Table 5.10: Framework Prototype lines of code

module	lines of code
core	1992
PaaS connector (CF)	834
strategy implementation (average)	39

To sum up, we have evaluated our approach from three aspects: SLA protection, robustness, and ease of use. We have conducted experiments to show: 1) how the DevOps team can use various strategies to conform to different SLA requirements. 2) how the framework can help the DevOps team to handle various faults through its kill-restart capability. Finally, we considered the required effort for using the proposed framework. As a result of our experiments, we believe that the proposed framework is easy to use for processing updates and programming strategies.

Chapter 6

Conclusion

Contents

6.1 Conclusions 86
6.2 Limitations and Future Works 88

6.1 Conclusions

Current agile development promotes to release software application updates at a daily frequency. With such shorter release cycles, DevOps teams get faster feedback from the customers, thereby reducing the risk of changes. However, processing updates at such level of frequency is challenging because any update is in itself a complex and tedious task. This is especially true when considering medium to large-size applications in which an update may impact many software components.

The microservices architecture helps reducing the complexity of processing software application updates, mainly because microservices split applications into services that can be updated independently from each others. But this advance is not enough to gain an easy-to-use way of updating applications, because to protect SLA (Service Level Agreement) properties when updating microservices, DevOps teams have to use complex and error-prone scripts of management operations.

Among the different strategies that are used to update microservices, the well-known BlueGreen strategy aims at updating a microservice with zero downtime, through deploying and starting all the new microservices before stopping and uninstalling the old ones. In comparison, the Canary strategy minimizes the resources used at update time, at the expense of a reduced availability: microservices are 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.

To follow strategies when updating their microservice applications, DevOps teams manipulate scripts composed of complex sequences of management operations. Management operations refer to the elementary operations exposed by the PaaS platforms (Platform as a Service) hosting microservices, allowing to control their lifecycle.

In addition to being complex, the scripts based on elementary PaaS operations are often specific to the type of change to process (code version, number of running instances, ...) and to the PaaS hosting the microservices to update.

The objective of the work proposed in this thesis is to make the work of the DevOps teams simpler and safer when these have to process updates. To this end, we propose an update framework that advocates switching from a script-based to an architecture-based approach to automate microservices updates: instead of scripts processing PaaS commands, update strategies are then defined as sequences of elementary changes being applied on an architectural model of a microservice application.

Such architecture-based approach allows to gain simplicity for the DevOps teams: a DevOps team can update a microservice application by only giving as input the desired target architecture along with the strategy to follow, without having to deal with low-level PaaS commands.

Strategies are defined through an architecture-based programming model, that make them easy to understand and easy to extend. Importantly, strategies are programmed independently of PaaS operations as well as independently of a particular microservice application. In other words, with our framework, a strategy describes a pattern of the update processes which is adaptable to various applications, change types, and PaaS platforms.

In addition to providing a higher level of automation, the proposed framework does not damage the flexibility of the update process. Thanks to its architecture-based character, our framework allows the DevOps team to execute an update process step by step, so that the DevOps team can easily integrate their proper actions during the update.

Finally, the proposed framework also helps the DevOps team to gain more safety in processing updates. First, the DevOps team can preview the path of intermediate architectures that will be followed to process a particular update, before effectively processing the update. Second, as previously said, the proposed framework provides a kill-restart capability. This means that the DevOps team can voluntarily or involuntarily stop the framework execution at any time during the update of a microservice application. Once stopped, the DevOps team can re-issue any update request to resume, rollback, or roll-forward the update of the application.

6.2 Limitations and Future Works

As a first point, we would like to say that although our experimentations have given encouraging results regarding the easiness of programming strategies, there are still work to do to finalize the conclusions regarding this aspect. In the next future, we plan to conduct a more complete analysis on the effort of strategy programming, considering a more large spectrum of microservice applications.

Regarding the limitations of our proposition, a main point that we only consider the update of microservices, not the update of the external services they may use (e.g., database, messaging, logging, or routing). Most PaaS solutions provide basic lifecycle operations (i.e., create, bind, unbind, and delete) to link microservices to external services through a broker API. With our framework, microservices that use external services may be updated without problem, but the update may only concern the microservices, not the external services they use. For example, our framework does not automate the update of a microservice along with upgrading its external database linked to the microservice. Supporting such updates constitutes a perspective of our work.

Regarding the other perspectives of our proposal, we would like to start with a main consideration: helping the DevOps team to choose the right strategy being given an application to update and the necessary SLA properties to maintain. For the future steps, we would like to investigate how the update framework could help the DevOps team to choose the most appropriate strategy for each particular update request.

To go one step in this direction, let's recall that our framework already enables the DevOps team to preview the path of intermediate architectures that will be followed by an update process. This preview functionality provides a basis to predict how well a strategy will protect SLA properties during the update. Especially, it is possible to identify in this path the moments where a microservice is fully stopped, the moments when a microservice run with less or more instances than before starting the update, etc.,.

Additionally, by instrumenting the different updates processed for a given application and by keeping persistent the instrumented data collected over time, we may predetermine more quantitative factors, such as the total downtime or the total resource costs of an update.

Finally, to conclude on this aspect, it may also be interesting to consider using formal techniques to precisely express the SLA constraints that have to be ensured during an update, as well as using verification techniques to predict whether a strategy can protect these invariants.

Bibliography

[1] Bluegreen update strategy. [Online]. Available: https://martinfowler.com/bliki/BlueGreenDeployment.html

[2] Canary update strategy. [Online]. Available: https://martinfowler.com/bliki/CanaryRelease.html

[3] F. Boyer, X. Etchevers, N. De Palma, and X. Tao, "Architecture-based automated updates of distributed microservices," in *International Conference on Service-Oriented Computing*. Springer, 2018, pp. 21–36.

[4] F. Boyer, X. Etchevers, N. de Palma, and X. Tao, "Poster: A declarative approach for updating distributed microservices," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 2018, pp. 392–393.

[5] T. Mauro, Adopting Microservices at Netflix: Lessons for Architectural Design. [Online]. Available: https://goo.gl/DyrtvI

[6] T. Hoff, Lessons Learned From Scaling Uber To 2000 Engineers, 1000 Services, and 8000 Git Repositories. [Online]. Available: https://goo.gl/1MRvoT

[7] Orange is migrating to microservices architecture. [Online]. Available: https://www.altoros.com/blog/orange-labs-test-massive-cloud-migration-with-elpaaso-add-on-to-cf/

[8] M. Amundsen, M. McLarty, R. Mitra, and I. Nadareishvili, *Microservice Architecture - Aligning Principles, Practices, and Culture*. O'Reilly Media, 2016.

[9] S. Fowler, *Production Ready Microservices*. O'Reilly, 2016.

[10] C. Carnero, *Microservices from day one: build robust and scalable software from the start*. Apress, 2016.

[11] A. Tarvo, P. F. Sweeney, N. Mitchell, V. Rajan, M. Arnold, and I. Baldini, “Canaryadvisor: A statistical-based tool for canary testing (demo),” in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. ACM, 2015, pp. 418–422. [Online]. Available: http://doi.acm.org/10.1145/2771783.2784770

[12] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*, 1st ed. Addison-Wesley Prof., 2010.

[13] A. R. Sampaio, H. Kadiyala, B. Hu, J. Steinbacher, T. Erwin, N. Rosa, I. Beschastnikh, and J. Rubin, “Supporting microservice evolution,” *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 539–543, 2017.

[14] N. Bencomo, R. France, B. Cheng, and U. Aßmann, *Models@run.time: Foundations, Applications, and Roadmaps.* springer, 01 2014, vol. 8378.

[15] B. W. Boehm, “A spiral model of software development and enhancement,” *Computer*, vol. 21, no. 5, pp. 61–72, 1988.

[16] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries *et al.*, “Manifesto for agile software development,” 2001.

[17] Martin Fowler’s article on Microservices. [Online]. Available: https://martinfowler.com/articles/microservices.html

[18] HashiCorp Consul Service Discovery. [Online]. Available: https://www.consul.io/

[19] Apache ZooKeeper. [Online]. Available: https://zookeeper.apache.org/

[20] Netflix Eureka. [Online]. Available: https://github.com/Netflix/eureka

[21] M. Nygard, “Release it!: Design and deploy production-ready software (pragmatic programmers),” *Pragmatic Bookshelf*, 2007.

[22] Netflix Hystrix. [Online]. Available: https://github.com/Netflix/Hystrix

[23] Twitter Finagle. [Online]. Available: https://twitter.github.io/finagle/

[24] Phantom. [Online]. Available: https://github.com/flipkart/phantom

[25] A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, and I. Stoica, "Above the clouds: A berkeley view of cloud computing," *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, vol. 28, no. 13, p. 2009, 2009.

[26] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs containerization to support paas," in *Cloud Engineering (IC2E), 2014 IEEE International Conference on.* IEEE, 2014, pp. 610–614.

[27] P. Mell and T. Grance, "The nist definition of cloud computing," *Communications of the ACM*, vol. 53, no. 6, p. 50, 2010.

[28] Cloud Service Levels. [Online]. Available: https://commons.wikimedia.org/wiki/File:Niveaux_de_service_cloud.png

[29] Amazon Elastic Compute Cloud. [Online]. Available: https://aws.amazon.com/ec2/

[30] Google Compute Engine. [Online]. Available: https://cloud.google.com/compute/

[31] OpenStack. [Online]. Available: https://www.openstack.org/

[32] Google App Engine. [Online]. Available: https://cloud.google.com/appengine/

[33] AppScale. [Online]. Available: https://www.appscale.com/

[34] OpenShift. [Online]. Available: https://www.openshift.com/

[35] Heroku. [Online]. Available: https://www.heroku.com/

[36] CloudFoundry. [Online]. Available: https://www.cloudfoundry.org/

[37] Google Kubernetes Engine. [Online]. Available: https://cloud.google.com/kubernetes-engine/

[38] Amazon EC2 Container Service. [Online]. Available: https://aws.amazon.com/ecs/

[39] Azure Container Service . [Online]. Available: https://azure.microsoft.com/en-us/services/container-service/

[40] Kubernetes. [Online]. Available: https://kubernetes.io/

[41] Apache Mesos. [Online]. Available: http://mesos.apache.org/

[42] Docker Swarm. [Online]. Available: https://docs.docker.com/engine/swarm/

[43] Martin Fowler's article on Deployment Pipeline. [Online]. Available: https://martinfowler.com/bliki/DeploymentPipeline.html

[44] Jenkins. [Online]. Available: https://jenkins.io/

[45] Concourse ci. [Online]. Available: https://concourse-ci.org/

[46] Gocd. [Online]. Available: https://www.gocd.org/

[47] M. E. Segal and O. Frieder, "On-the-fly program modification: Systems for dynamic updating," *IEEE software*, no. 2, pp. 53–65, 1993.

[48] S. Subramanian, M. Hicks, and K. S. McKinley, "Dynamic Software Updates: A VM-centric Approach," in *PLDI'09: Proc. of the ACM SIGPLAN conf. on Programming Language Design and Implementation*, Dublin, Ireland, 2009, pp. 1–12.

[49] A. Gregersen, M. Rasmussen, and B. Jørgensen, "Dynamic software updating with gosh! - current status and the road ahead." in *ICSOFT*, J. Cordeiro, D. A. Marca, and M. van Sinderen, Eds. SciTePress, 2013, pp. 220–226. [Online]. Available: http://dblp.uni-trier.de/db/conf/icsoft/icsoft2013.html#GregersenRJ13

[50] M. Pukall, A. Grebhahn, R. Schröter, C. Kästner, W. Cazzola, and S. Götz, "Javadaptor: Unrestricted dynamic software updates for java," in *Proc. of the 33rd Int. Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 989–991. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985970

[51] OSGi Service Platform Core Specification, Release 5, Last retrieved July 2016., http://www.osgi.org/Specifications/HomePage. [Online]. Available: http://www.osgi.org/Specifications/HomePage

[52] Springcloud. official website. [Online]. Available: http://projects.spring.io/spring-cloud/

[53] Eclipse, "Eclipse RCP, http://www.eclipse.org, 2008."

[54] J. Kramer and J. Magee, "The Evolving Philosophers Problem: Dynamic Change Management," *IEEE TSE*, vol. 16, no. 11, pp. 1293–1306, 1990.

[55] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The fractal component model and its support in java," *Softw., Pract. Exper.*, vol. 36, no. 11-12, pp. 1257–1284, 2006.

[56] F. Boyer, O. Gruber, and D. Pous, "Robust reconfigurations of component assemblies," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds. IEEE Computer Society, 2013, pp. 13–22. [Online]. Available: https://doi.org/10.1109/ICSE.2013.6606547

[57] B. Morin, O. Barais, G. Nain, and J.-M. Jezequel, "Taming dynamically adaptive systems using models and aspects," in *Proc. of the 31st IEEE Int. Conf. on Software Engineering (ICSE'09)*, Washington, DC, USA, 2009, pp. 122–132.

[58] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan, "A generic component model for building systems software," *ACM Transaction on Computer Systems*, vol. 26, no. 1, pp. 1–42, 2008.

[59] J. Armstrong, *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2013.

[60] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala: Updated for Scala 2.12*, 3rd ed. USA: Artima Incorporation, 2016.

[61] J. Allen, *Effective Akka*. O'Reilly Media, Inc., 2013.

[62] Spinnaker pipeline parameters. [Online]. Available: https://www.spinnaker.io/guides/user/pipeline-expressions/#pipeline-parameters

[63] Spinnaker developing feature pipeline templates. [Online]. Available: https://github.com/spinnaker/pipeline-templates

[64] R. Di Cosmo, A. Eiche, J. Mauro, S. Zacchiroli, G. Zavattaro, and J. Zwolakowski, "Automatic deployment of services in the cloud with aeolus blender," in *Proceedings of the 13'th International Conference on Service-Oriented Computing*, ser. ISOC '15, 2015, pp. 397–411.

[65] M. Gabbrielli, S. Giallorenzo, C. Guidi, J. Mauro, and F. Montesi, "Self-reconfiguring microservices," in *Essays Dedicated to Frank De Boer on Theory and Practice of Formal Methods - Volume 9660*. New York, NY, USA: Springer-Verlag New York, Inc., 2016, pp. 194–210. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-30734-3_14

[66] Jolie. Official Web Site. [Online]. Available: http://www.jolie-lang.org/

[67] T. A. Lascu, J. Mauro, and G. Zavattaro, "A planning tool supporting the deployment of cloud applications," in *Proceedings of the 2013 IEEE 25th International Conference on Tools with Artificial*

Intelligence, ser. ICTAI '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 213–220. [Online]. Available: http://dx.doi.org/10.1109/ICTAI.2013.41

[68] J. Fischer, R. Majumdar, and S. Esmaeilsabzali, "Engage: A deployment management system," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 263–274. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254096

[69] X. Etchevers, T. Coupaye, F. Boyer, N. D. Palma, and G. Salaün, "Automated Configuration of Legacy Applications in the Cloud," in *Proc. of UCC'11*. IEEE Computer Society, 2011, pp. 170–177.

[70] R. Di Cosmo, M. Lienhardt, J. Mauro, S. Zacchiroli, G. Zavattaro, and J. Zwolakowski, "Automatic Application Deployment in the Cloud: from Practice to Theory and Back ," in *Proceedings of 26th International Conference on Concurrency Theory (CONCUR 2015)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 42. Madrid, Spain: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Sep. 2015, pp. 1–16. [Online]. Available: https://hal.inria.fr/hal-01233426

[71] J. Martin, *Managing the Data Base Environment*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1983.

[72] Open-source prototype of the proposed DMU framework. [Online]. Available: https://github.com/tao-xinxiu/prototype-template-engine

[73] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[74] A. Avižienis, J.-C. Laprie, and B. Randell, "Dependability and its threats: a taxonomy," in *Building the Information Society*. Springer, 2004, pp. 91–120.

[75] J. Shore, "Fail fast [software debugging]," *IEEE Software*, vol. 21, no. 5, pp. 21–25, 2004.

[76] Cloudwatt. The public Orange cloud. [Online]. Available: https://www.cloudwatt.com

[77] Open-source microservices application Account. [Online]. Available: https://github.com/paulc4/microservices-demo

[78] The architecture of the open-source microservices application Account. [Online]. Available: https://github.com/paulc4/microservices-demo/blob/master/mini-system.jpg

[79] Too many instance replacements can cause a cascading failure on Cloud Foundry. [Online]. Available: https://docs.cloudfoundry.org/adminguide/diego-cell-upgrade.html