Towards Supporting the Extended DevOps Approach through Multi-cloud Architectural Patterns for Design and Pre-deployment
A Tool Supported Approach

Juncal Alonso[1], Marisa Escalante[1], Lena Farid[2], Maria Jose Lopez[1], Leire Orue-Echevarria[1] and Simon Dutkowski[2]

[1]*ICT Divsion, TECNALIA, Bizkaia Technology Park, Derio, Spain*

[2]*Fraunhofer Institute for Open Communication Systems (FOKUS), Berlin, Germany*

Keywords: Cloud Computing, Multi-cloud, Continuous Design, Continuous Pre-deployment, Deployment Optimization, DevOps.

Abstract: Recently the world of Cloud Computing is witnessing two major trends: Multi-cloud applications pushed by the increasing diversity of Cloud services leading to hybrid infrastructures and the DevOps paradigm, promising increased trust, faster software releases, and the ability to solve critical issues quickly (Steinborn, 2018). This paper presents a solution for merging and adapting both trends so that the benefits for software developers and operators are multiplied. The authors describe a tool-supported approach to extend the DevOps philosophy with the objective of supporting the design and pre-deployment of multi-cloud software applications. The paper begins with the presentation of the theoretical concepts, the proceeds with the description of the developed tools and the discussion of the validation performed with a sandbox application.

1 INTRODUCTION

In the past few years, DevOps has become a common word. DevOps is a set of practices that automates the processes between software development (Dev) and IT teams (Ops) so they can build, test, release and deploy software applications more quickly, reliably and continuously. In traditional DevOps approaches, IT roles are merged and communication is enhanced to improve the production release frequency and maintain software quality (Riungu-Kalliosaari et al., 2016). The foreseen benefits of the DevOps philosophy include increased trust, faster software releases, automated testing and the ability to solve critical issues quickly.

In parallel, the world of cloud computing is witnessing a growing trend by the increasing diversity of cloud services offerings leading to hybrid and multi-cloud infrastructures, which are available for complex software applications that can profit from these variety of offers and features.

Multi-cloud can be understood as the use of multiple computing services for the deployment of a single application or service across different cloud technologies and/or Cloud Service Providers. This may consist of PaaS, IaaS and SaaS entities (Gavilanes et al., 2017). This definition of multi-cloud, when referring to the resources where different components are deployed, includes services which are in disperse cloud providers or different cloud platforms (regardless of vendor) (Escalante et al., 2017)

Following this definition, the authors understand “multi-cloud” based applications, as software applications that are defined as a set of components distributed across heterogeneous cloud resources but that still succeed in interoperating as a single whole.

Developing, deploying and operating such multi-cloud applications present challenging shortcomings, that have not been deeply analysed nor supported by current DevOps solutions. These multi-cloud specific peculiarities include:

- Applications need to be responsive to hybrid/multi-cloud model scenario in which an application that is executing in a concrete set of cloud services bursts into a new one when the working conditions are not met. This implies that the application architecture shall be re-designed to be “multi-cloud” aware simplifying the cloud application assembly and the deployment process.

Alonso, J., Escalante, M., Farid, L., Lopez, M., Orue-Echevarria, L. and Dutkowski, S.
Towards Supporting the Extended DevOps Approach through Multi-cloud Architectural Patterns for Design and Pre-deployment - A Tool Supported Approach.
DOI: 10.5220/0006856008130823
In *Proceedings of the 13th International Conference on Software Technologies (ICSOFT 2018)*, pages 813-823
ISBN: 978-989-758-320-9
Copyright © 2018 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

- Means shall be provided to manage and assess cloud deployment alternatives to better support cloud re-deployment decisions. This implies profiling and classifying application components and cloud nodes, as well as analysing and simulating the behaviour of the application to support the deployment decision making process considering additional factors such as Non-Functional Requirements (NFR), namely performance, availability, localization, cost, or risks associated with the change of cloud resources. Multi-cloud has value only when the right providers are selected, whether public or private (combined into different cloud deployment models), to meet functional and NFR. But the manual selection and combination of those Cloud Services to create the best deployment scenario may imply huge effort, time and knowledge needs.
- Existing cloud services shall be made available dynamically, broadly and cross border. so that software providers can re-use and combine cloud services, assembling a dynamic and re-configurable network of interoperable, legally secured, quality assessed (against SLAs) single and composite cloud services,

The current paper presents a solution proposed by the authors extending the DevOps approach with the objective of supporting the design and pre-deployment of multi-cloud software applications, while facing the aforementioned challenges, imposed by the peculiarities of the multi-cloud deployment.

2 EXTENDED DEVOPS APPROACH FOR MULTI-CLOUD APPLICATIONS

2.1 Traditional DevOps

DevOps refers (DECIDE Consortium, 2017) to the emerging professional movement and philosophy that advocates for a collaborative working relationship between Development and IT Operations, lowering barriers and silo-based teams, resulting in the fast flow of planned work (i.e., high deploy rates, better quality and faster releases), while simultaneously increasing the reliability, stability, and resilience of the production environment. This is often called the "DevOps Paradox" (Edwards, 2015): "*Going faster brings higher quality, lower costs, and better outcomes*". DevOps pivots around three axes (UpGuard, 2016): processes, people and technology. From the people perspective, DevOps symbolizes a cultural change where collaboration and cooperation are key pillars, and this often results in an increased understanding to prioritize requests that the business needs. From the processes perspective, DevOps advocates for more agile change processes, with an increased rate of acceptance for new features, improved quality in software developments, a decrease in number of incidents per release and an increased time to market and velocity to pass from development to production. Finally, from the technology point of view, DevOps results in an application with a reduced number of defects and therefore with more quality, and in an increased deployment of features.

After an analysis of the current coverage of the DevOps approach to the peculiarities of multi-cloud applications and considering the phases of the software development lifecycle (SDLC), and software operation lifecycle (SOLC) of these targeted applications, the authors propose a novel extended DevOps approach to overcome current shortcomings.

2.2 Extended DevOps = DevOps + Continuous Architecting + Pre-deployment

The work presented here is being developed in the context of the DECIDE project, that proposes an extension of the "traditional" DevOps approach on both axis: Dev and Ops. This paper, however, presents an approach for the extension of the Development phase. For the Dev axis, the authors propose to extend the development phase with previous and subsequent activities that cover: 1) the architectural design for multi-cloud applications (at generic level and specific to non-functional requirements), 2) the election of the best deployment configuration based on theoretical simulations. With this in mind, the authors propose an "Extended Dev" covering:

- Multi-cloud applications continuous architecting and development
- Multi-cloud applications (pre) deployment

2.2.1 Existing Approaches and Tools

There are some on-going initiatives to create a DevOps framework integrating development and operation tools into a unique framework.

The Eclipse Cloud Development project (Eclipse Cloud Development, 2018), composed of four sub-projects (Che, Orion, Dirigible and Flux), is aiming

for a rich IDE to be executed through the browser. This could be considered as DevOps but it does not provide other advanced features. IBM Bluemix DevOps Services (IBM BlueMix, 2016) can be used by a web-based IDE or plug-ins for the most common IDEs. It allows the automatic build and deployment to IBMs cloud platform Bluemix, built on top of the PaaS engine CloudFoundry. BlueMix also covers continuous integration mechanisms as the update of deployed applications. None of these tools, cover the peculiarities of the multi-cloud applications at architectural level, nor analyse the impact of non-functional properties in the design of the multi-cloud application and vice-versa.

For the deployment simulation, the authors have not found a similar tool.

However, most of these initiatives, open source projects and commercial tools do not fully support the SDLC and SOLC of a multi-cloud application as they target one cloud platform at a time, and also miss other aspects related with the architectural aspects of the applications or the definition of the “best” deployment configurations (DECIDE Consortium, 2017).

At research level, it has been established in different works, that the adoption of DevOps and/or Continuous Delivery (CD) brings new challenges to software development and thus affects the design of applications. There are numerous works dedicated to this topic. Some of which, deal with architecting for CD in terms of tooling (Mojtaba, 2015) and compliance with Architecturally Significant Requirements (ASR) (Chen, 2015). These approaches are meaningful and very much needed as they have proven to facilitate the deployability, modifiability, security, monitorability, loggability, and testability of applications (Chen, 2015).

However, when it comes to Cloud technologies, especially multi-cloud, other tools, requirements and best practices for Cloud native applications are needed on top of the aforementioned because herein additional architectural challenges are permitted.

There has been little research in the multi-cloud domain targeted at the design process in conjunction with the pre-deployment of (and ultimately DevOps/CD) for multi-cloud native applications, as it is a fairly new field. The existing models and systems are still maturing.

Nevertheless, tools for architectural best practices or patterns for cloud native applications have been researched in the ARTIST project (Kopanelli et al., 2015) and MODAClouds (Di Nitto et al., 2017).

The ARTIST project produced a refined list of Cloud Computing optimization patterns (ARTIST Consortium, 2014) for migrating legacy applications into the Cloud enhancing the non-functional properties and MODAClouds produced a set of concepts and patterns targeted at multi-cloud. MODAClouds additionally looked at DevOps tooling for application design, cloud services selection and automated deployment. Another point that differs these two projects from the authors’ goals is that they considered an application as a whole in contrast to investigating a decomposed application and each component’s individual requirements and business needs.

2.2.2 Challenges

In the following table, the authors summarize the challenges encountered in each of the phases and sub-phases of the traditional DevOps approach, when applying it to multi-cloud applications. Some of the challenges encountered have served as motivation for the proposal of the tools and mechanisms exposed in section 3.

Table 1: Challenges identified at “Dev” phase.

Sub-phase	Challenge
Design	NFR specification
Design	Best practices for multi-cloud architectural design
Implementation	Multi-cloud Development support
Integration	Integration of tools in the Dev phase
Pre-deployment	Application (nodes and communication included) profiling/classification
Pre-deployment	Automatic Theoretical deployment generation
Pre-deployment	CSP modeling
Pre-deployment	Simulation (deployment)
Pre-deployment	Automatic Cloud services discovery
Optimization	Code optimization
Design/Pre-deployment	(MC)SLA definition

In this paper, the Multi Cloud Service Level Agreement (MCSLA) definition and the proposed solution on how to approach this challenge in the context of the multi-cloud applications Dev phase has not been described.

3 TOOLS AND MECHANISMS SUPPORTING THE DESIGN AND PRE-DEPLOYMENT OF MULTI-CLOUD APPLICATIONS

As presented in the previous section, the extension of

the Dev phase introduces new activities to be carried out by the developers of multi-cloud applications, both before and after the actual development phase takes place, which currently are not supported by existing tools. The DECIDE project aims to implement several tools which support these new activities incorporated to the DevOps cycle. In the following section, the functional description, structural architecture and technology used for the implementation of these supporting components are explained.

3.1 ARCHITECT: Continuous Architecting

Continuous architecting is a step introduced in order to support the design process in an application's lifecycle with the motivation of moving back and forth within said lifecycle in order to satisfy the needs of other DevOps phases, such as continuous pre-deployment, continuous deployment, monitoring, etc. in a multi-cloud context.

Furthermore, when integrating the design process into DevOps, the adoption of business and technical requirements (i.e. NFRs or deployment targets) becomes possible at a very early stage of the applications lifecycle and thus, any defined requirement from Dev side is easily injected into each Ops task (from CI/CT to pre-deployment, through deployment and monitoring).

In light of the novelty and peculiarities of multi-cloud deployment and operation, developers and organisations may not be familiar with applying the best architectural solutions for complying in such an environment. Therefore, the authors propose a tool as part of the continuous architecting phase which incorporates a large number of best practices, namely multi-cloud architectural patterns. This will aid developers in tackling multi-cloud idiosyncrasies.

3.1.1 Multi-cloud Architectural Patterns

A design or architectural pattern provides a general, reusable solution to a commonly occurring problem in the development and/or deployment of software components (Gamma, et al., 1995). The solutions are provided as descriptions and are never a fully finished design. This is intentional in order for these patterns to be applicable to a wide range of scenarios and remain vendor agnostic.

In view of this, the use of architectural patterns in object-oriented programming and distributed applications has dramatically improved many aspects in software and systems engineering, such as their quality, speed maintainability and accessibility. For the same reason, a large number of Cloud Computing architectural patterns have been developed (Fehling, et al., 2014).

Since the focus of the DECIDE project is to aid developers in designing multi-cloud aware applications (and not single-cloud aware), the authors have curated a catalogue of multi-cloud architectural patterns consisting of a number of cloud computing patterns for distributed applications (and presumably non-cloud patterns). The patterns collectively address the multi-cloud peculiarities. The patterns have been categorised as *fundamental*, *development*, *deployment* and *optimisation* patterns (Farid, et al., 2017).

Fundamental patterns are those deemed necessary for the use of the DECIDE DevOps Framework and provide a minimal set that needs to be applied. An example of such pattern would be to containerize the application.

Development patterns are those that aid the developer with best practices for building a multi-cloud application. Example patterns are: distributed application (splitting the application into microservices), loose coupling or stateless.

Deployment patterns address how the deployment configuration for multi-cloud applications should be handled. For instance, managing the deployment scripts as well as storing them should be designed from a multi-cloud perspective. Here types of technological risks as well as geographical locations of the components (data or business logic) are accounted for. Furthermore, the deployment patterns take into account DECIDE principles of re-adaptability and re-deployment for multi-cloud environments.

Optimization patterns are those that aid the developer in improving the applications NFRs by taking adequate measures in optimizing the application code to reflect on these requirements. Optimization patterns can optimize the use of cloud resources, such as elasticity. An example is leveraging a cloud persistence layer instead of implementing it as part of the application.

In the context of DECIDE, these patterns will be fed into the ARCHITECT tool. ARCHITECT will then use different information about the application along with the NFRs to recommend multi-cloud architectural patterns for the developers.

3.1.2 The ARCHITECT Tool

The ARCHITECT tool holds a catalogue of architectural patterns and supports the developer with

preparing the application for a multi-cloud deployment scenario by recommending a set of (multi-)cloud architectural patterns, which must or should be applied to the application.

By preparing the application for a multi-cloud deployment scenario, the authors mean optimising an application's development and deployment in order for an application to be dynamically re-adapted and re-deployed.

The patterns recommendation is based on the selected and prioritized NFRs as well as on additional data concerning the application and its components. The recommended patterns provide a description of how these patterns can and may be applied to the source code. The tool is designed to be close to the development process, yet is language agnostic and can be run as a RESTful microservice.

Structural Description

The ARCHITECT component has a set of functional requirements that can be summed up into the following functionalities:

- Provide/ recommend users (i.e. the developer) architectural patterns based on their prioritized NFRs as well as additional information (supplied by the users), with guidelines on how to apply them, to which component (e.g. microservice) it needs to be applied and in which order.
- Provide a repository of multi-cloud architectural patterns.

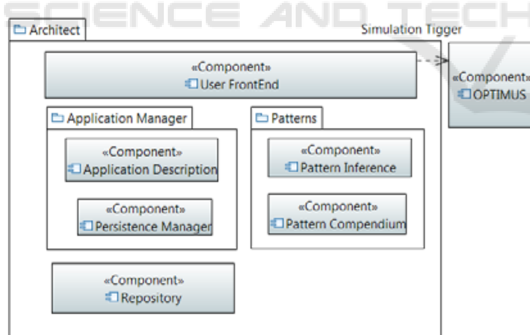

Figure 1: Component Diagram of ARCHITECT.

ARCHITECT is decomposed in several functional blocks and interfaces. It consists of three core elements, as depicted in the figure above. A frontend for user interaction, the application description manager for dealing with the DECIDE project model, and finally the patterns catalogue with the pattern inference engine.

- User Frontend. This element depends on the context. For example, if ARCHITECT is integrated in an IDE or is a web based solution, this part provides the mechanism for plugging in the ARCHITECT component. Its main task is the interaction with the developer and it provides the necessary user interfaces to view and manage the patterns and also to trigger the recommendation process. The User Frontend is the workflow-controlling component of ARCHITECT.
- Application Manager. This element is responsible for a convenient abstraction level for the information model of the DECIDE application. It manages all application information in a persistent manner. That means, it encapsulates and hides the technical details, (e.g. the fact that the application description is coded and stored as a JSON structure inside a code repository).
- Patterns. This element contains a catalogue of patterns, NFRs and their relationships. The contained information can be enriched to hold additional information experienced over time. The patterns catalogue provides functions that allow the inferring of patterns based on a given set of NFRs and optionally some fixed patterns.

Technical Description

ARCHITECT's Patterns component is implemented as an autonomous Java library and its functionality is offered also as a RESTful microservice and as such is accessible for other implementations. This allows an easy integration of ARCHITECT in a polyglot environment

For instance, its functionality can be integrated via web-based tools by using its API or directly into a Java client, such as an IDE (e.g. Eclipse).

The first attempt for the pattern recommendation is realised using semantic technologies. The patterns are semantically enriched by using ontology based techniques and rely on the NFRs for inferring viable patterns.

The means for communicating with the pre-deployment phase (OPTIMUS tool) is realised using a common DevOps mechanism. The recommended patterns, which are relevant for the pre-deployment phase, are stored in the Application Description, a JSON file in a git repository. This file acts as a single point of truth for the multi-cloud application and may be easily managed and accessed via the Application Manager library.

The Application Manager library encapsulates all git operations for reading and writing to the Application Description JSON file (Farid et al., 2017).

3.2 OPTIMUS: Continuous Simulation

DECIDE OPTIMUS deployment simulation tool eva-

luates and optimizes the characteristics of the multi-cloud application deployment from the developer's perspective considering a set of provided cloud resources alternatives.

DECIDE OPTIMUS provides the best possible deployment application topology, based on the non-functional requirements set by the developer and the requirements of the multi-cloud application, automating the provisioning and selection of deployment schemas for multi-cloud applications.

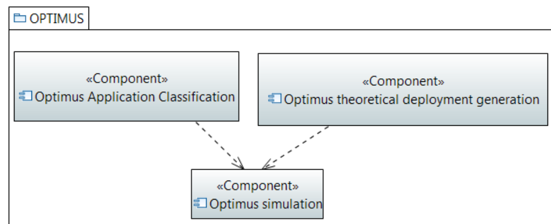

Figure 2: OPTIMUS High level architecture.

3.2.1 Structural Description

DECIDE OPTIMUS consists of three main components and each of them provides the corresponding functionality:

- Application Classification

The Application Classification subcomponent interacts with the developer through OPTIMUS UI, from where the developer provides the information about the multi-cloud application that OPTIMUS needs to classify each of the microservices of the multi-cloud application. The Application Classification process will match the information stored about the types of microservices and the characteristics associated with each of the multi-cloud application microservices.

The concept "types of microservices" can be found in the OPTIMUS architecture and in the related references to the classification process. Each of the microservices need a Cloud Service that fulfils the requirements for deploying it properly. Type of microservices groups a set of characteristics that are required from a Cloud Service to be selected and matched with a specific microservice.

The output of the classification is stored into the Apps Classification Repository, physically located in the Application Description JSON file.

The Types management subcomponent manages the system knowledge about the Cloud Services that the microservices need to be deployed.

- Theoretical deployment generation

Once the classification is completed, knowing the type of each microservice, OPTIMUS is capable to work out the Cloud Services that are suitable for it. The Deployment Types Repo Management performs the equivalence among the microservices types and the CSPs in which they could be theoretically deployed.

Considering the classification, the characteristics of each microservice and the NFRs associated to them by the developer, the Theoretical Deployment Preparation creates a list of requirements to invoke to the Advanced Cloud Service (meta-) intermediator (ACSmI) Discovery Service and obtain the Cloud Services that meet them.

This request is composed of generic Cloud Services, and the list of resources that the microservices need. This functionality requires interacting with ACSmI's API to obtain the sorted list of Cloud Services that meet the criteria requested.

- Simulation

Keeping in mind the obtained group of different possibilities for the deployment, a complex algorithm performs a combination of all these possibilities and scores them in a list. The first theoretical deployment in that list will be the "best schema", and if this schema has not been selected before (checking the historical deployment configurations repository), it is presented to the developer.

Within the continuous approach of DECIDE OPTIMUS tool, this Simulation phase can be triggered the first time when the application is deployed, by the developer, or it can be triggered when a violation of the MCSLA occurs. In this case, the current deployment configuration is invalidated and OPTIMUS should obtain another best schema for the deployment, considering the new circumstances.

3.2.2 Technical Description

The Classification module and the UI for collecting the information about the microservices, has been developed as an eclipse plugin using Eclipse Java EE IDE for Web Developers, Version Oxygen.1 Release (4.7.1). The simulation process is a RESTful service that could be invoked from different points of the DECIDE framework.

3.3 ACSmI: Continuous Resource Discovery

The Advanced Cloud Service (meta-) intermediator (ACSmI) (Escalante et al., 2017) aims to provide the means for the discovery, contracting, managing and monitoring of different cloud service offerings. In this paper, the authors explained the approach followed by the ACSmI discovery functionality.

As authors mentioned in (Escalante et al., 2017), ACSmI Discovery component covers the functionality of services discovery based on the request passed by OPTIMUS. To be able to discover services, these services should be endorsed previously in the service registry. The endorse functionality, as well as, the modification and deletion of the services in the registry are also covered by this component. The most relevant requirements covered in this component are:

- Collect the requirements of OPTIMUS, these requirements are specified following the different terms defined for the modelling of the CSPs and their services. This allows for an automatic comparison of the requirements with the services stored in the registry.
- Provide a list of services from the service registry that fulfil (totally or partially) the requirements specified by OPTIMUS.
- Provide means to create, read, update and delete the services registry.
- To allow CSPs or ACSmI administrator (for Large CSPs) to endorse their services in the service registry. The registry of each service covers the different terms defined in the modelling of the CSPs and their services. This allows the automatic discovery of the services from the registry.

3.3.1 Structural Description

ACSmI discovery component is structured in four main components, as shown in Figure 3.

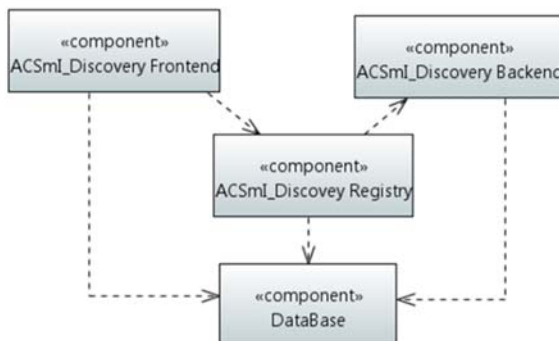

Figure 3: ACSmI Discovery Components.

- Frontend. This component has two main objectives: 1) To implement the graphical interface to allow inserting the requirements for the discovery of the services as well as to allow CSPs to introduce the information regarding their services, 2) To manage users. This component enables to create, read, update and delete users in ACSmI discovery.
- Backend. This component is in charge of 1) carrying out the discovery of the services based on the information provided by the user and 2) the endorsement of the services based on the information provided by the CSP through the frontend. To carry out these activities, this component manages the database to create, delete, modify the service types, services, CSPs, etc.
- Registry. This component coordinates the communication between the frontend and the backend.
- Database, which stores the services. This component is a MySQL database, following the data model shown in the Figure 4.

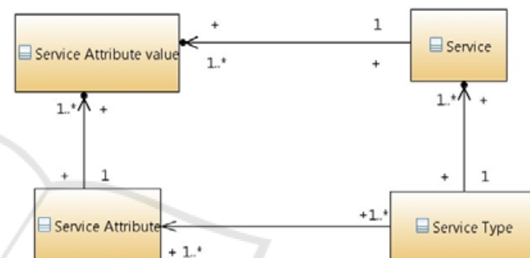

Figure 4: ACSmI Cloud services model.

3.3.2 Technical Description

ACSmI Discovery has been developed using JHipster. The main objective of JHipster (JHipster, 2015) is to generate a modern and complete web application or microservice architecture. JHipster is not a development framework, it is more a frontend and backend generator based on different technologies. JHipster allows the use of Swagger to define the communication between the frontend and backend. Each functional component communicates with the other through REST API, even though these components are in the same service of the backend.

4 THE SOCKSHOP MICRO-SERVICE BASED APPLICATION: INITIAL APPROACH VALIDATION

The SockShop App (Weaveworks, 2016) has been selected as an exemplary application to showcases which multi-cloud patterns can be applied in order to render it multi-cloud aware and which Cloud Services fit best in a multi-cloud scenario.

The different components developed and used for the validation of the proposed approach (ARCHITECT, OPTIMUS, ACSmI) are still isolated components (not integrated) so that the results obtained through the exercise performed with the SockShop application within the three components are not linked. In the next versions, the validation will be done with the integrated components so that the results obtained through the different components are coherent.

Again, multi-cloud aware in the context of the DECIDE project implies that the application is distributed over multiple heterogeneous CSPs and can be seamlessly re-deployed, i.e. ported across CSPs and re-adapted.

4.1 SockShop Architecture

The selected application is a loosely coupled microservices demo application. It simulates the user-facing part of an e-commerce website that sells socks. It is open source software and has been developed with the intention to aid in demonstrating and testing microservices and cloud native technologies (Weaveworks, 2016).

With this in mind, the SockShop App is designed to provide as many microservices as possible. The microservices are designed to encapsulate functionality required in an e-commerce site and are, of course, loosely coupled. The microservices are designed to have minimal expectations and use DNS to find other services. The Application uses a message broker for sending messages by means of queues (Weaveworks, 2016). All services communicate using REST over HTTP. Furthermore, the SockShop App is polyglot as it is built using Spring Boot (Spring, n.d.), Go kit (Bourgon, 2017) and Node.js (Node.js Foundation, n.d.) and is packaged in Docker (Docker, n.d.) containers. Figure 5 shows the original architecture of the application.

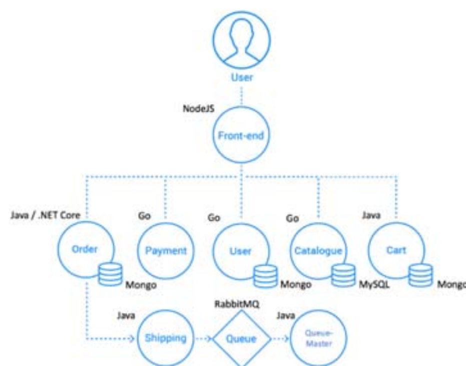

Figure 5: SockShop Application Architecture (Weaveworks, 2016).

4.2 Preparing for Extended DevOps

For the SockShop App a set of NFRs have been defined, based on hypothetical assumptions in the context of an e-commerce application and prioritize them as follows:

- Scalability – Hypothetically, it is assumed that market research and data analytics show that the user base is active during morning hours and after 8 PM otherwise unpredictable workloads can be expected.
- Performance – the performance of the application is important as the users expect rendering of the website and the transactions speed to be at most 2 seconds.
- Availability – To maintain a good reputation the service has to be available at 99% or at all times.
- Cost – As the SockShop is a start-up, keeping costs to a minimum is vital.

With this list of NFRs set, the pattern recommendation can commence with regards to preparing the application for a multi-cloud environment.

4.3 Candidate DECIDE Patterns

As understood by the wider software engineering community, architectural patterns provide solutions or best practices for commonly occurring problems (Fehling et al., 2014). With a pattern-based approach, developers can be guided in preparing their applications for a multi-cloud environment and knowledge can be acquired with regards to the most optimal deployment topology.

This section introduces a selection of design patterns that address the NFRs and the original architecture, Furthermore, insight is given with regards to their relevance for the SockShop App.

DECIDE Fundamental Patterns
The SockShop App fulfils evidently a number of patterns that are fundamental for the use of the DECIDE Framework. These are:

- Distributed Application - The SockShop App consists of microservices. This allows the application to be deployed in a distributed manner and thus make use of a multi-cloud strategy.
- Loose Coupling - The microservices communicate via REST over HTTP.
- Three-Tier Cloud Application - Front-end, Business Logic (Order, Shipping, Payment), Persistence (Order, User, Catalogue, Cart)
- Containerization (Farid et al., 2017) – The Sock-

Shop App is developed as container-based architecture.

With these patterns the ground work for the NFRs: **Scalability, Performance, Cost and Availability** can start to be addressed.

Other fundamental patterns that still need to be applied are:

- Managed Configuration – Deployment and configuration scripts have to be stored in a central area external to the built binaries.
- Service Registry - The SockShop App uses DNS to discover services. As DNS propagation is slow, using DNS tables is probably problematic in a multi-cloud scenario, because of re-deployment and access issues. Furthermore, given the fact, that many instances will be spawned or scaled out and there is probably a number of communications taking place between the different microservices, this needs to be handled by a service. Therefore, the authors propose the service registry pattern, with which a type of database or table dynamically holds the current location of the services, their instances and locations. Registration and de-registration of the service instances takes place during start-up and shutdown, respectively.

DECIDE Optimization Patterns

- Elastic Load Balancer – As workloads are unpredictable at certain times (during the day) it is vital to scale out automatically depending on the current experienced workload. The components resulting in being scaled out by an elastic load balancer are Front-End, Order, Payment, User, Catalogue and Cart.
- Elastic Queue – since the SockShop App uses message queues in its architecture and scalability is an important NFR, an Elastic Queue should be employed to manage the number of instances (Shipping and QueueMaster) depending on the number requests to be queued.

DECIDE Development Patterns

The SockShop App fulfils a number of development patterns that are part of the DECIDE multi-cloud pattern catalogue. These are:

- Data Access Component – The microservices, which access a data base are themselves implemented in a way that isolates complexity of data, enable additional data consistency, and ensure adjustability of handled data elements to meet different customer requirements.
- User Interface Component – The front-end microservice is decoupled from the rest of the application (i.e. microservices) and loosely coupled. The front-end is therefore, exchangeable and customisable. Furthermore, it can be scaled out independently if need be.
- Processing Component – The SockShop App's microservices can be scaled out independently, as separation of concerns has been considered here at the design time of the application.

DECIDE Deployment Patterns

- Hybrid * - The patterns involving hybrid cloud, such as hybrid user interface, hybrid processing, hybrid data, hybrid backup, hybrid backend, and hybrid application functions (Fehling et al., 2014) all involve using multiple hosting environments that best suit the requirements and needs of the application. This is relevant in a multi-cloud strategy and can drastically reduce cost if, for instance, certain microservices do not require elasticity they can be hosted on a private cloud that does not feature these capabilities. Also sharing IT-resources between different tenants can drastically reduce costs.

4.4 Resulting Architecture

Figure 6 depicts the architecture for the SockShop App after the recommended patterns have been applied. As one can see, an independent **Configuration Manager** component has been introduced to allow for dynamic configuration of the microservices as well as allow other automated deployment and provisioning tools to access the configuration information needed for their tasks.

Furthermore, a **Service Registry** has been introduced in order to facilitate the discovery of the location of the microservices that have been newly instantiated by the **Elastic Load Balancer**. And lastly an **Elastic Queue** manages the number of needed Queue Masters depending on the number of the messages received by the Rabbit MQ.

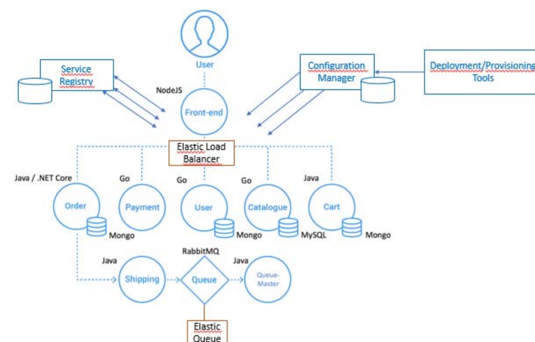

Figure 6: Resulting Architecture of SockShop Application.

4.5 OPTIMUS. Classification

Once the developer knows which are the different microservices of the application, he can introduce information related to them through the Classification User Interface.

The collected data include the nature of the microservice (developed by the user or not), if it is Stateless, if it needs a IP to expose it publicly to access to it, the dependencies among the rest of the microservices (order), if it uses a detachable resource and its characteristics, and the Non- Functional requirements.

Finally, when the information about all microservices is completed, based on the groups of Cloud Services managed by the tool, the result of the classification for the SockShop Application would be as presented in the next section.

4.6 OPTIMUS. Simulation

The simulation process, based on the previous classification made, includes as a first step, the association among each microservice and the group of Cloud Services that can be suitable for their deployments, as is depicted in the table 2.

Table 2: Cloud Services classification for the SockShop Application.

Microservice	Type of Cloud Service needed
Front-end	VM or Container (with Public IP)
Order	VM or Container + DB
Payment	VM or Container.
User	VM or Container + DB
Catalogue	VM or Container + DB
Cart	VM or Container + DB
Shipping	VM or Container.
QueueMaster	VM or Container + QueueSystem

The NFRs are very important aspects in order to be able to select the best deployment within the DECIDE framework. Considering these NFRs, assigned by the developer, and the group of Cloud Services selected in the Classification phase, OPTIMUS creates a specific request to invoke to the ACSmI Discovery Service.

Considering location as one of NFRs selected and the value set to Ireland, OPTIMUS could build the structure of the request to perform, for the two first microservices.

The service exposed by ACSmI Discovery will be invoked including the prior mentioned request, and as a result, OPTIMUS would receive a list of Cloud Services that fulfil the requirements.

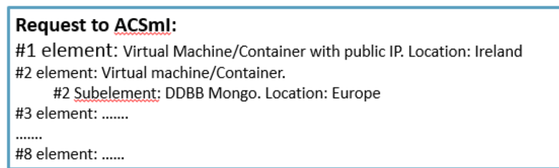

Figure 7: Request to invoke to ACSmI Discovery.

Figure 8 shows the ACSmI Discovery response for the request made.

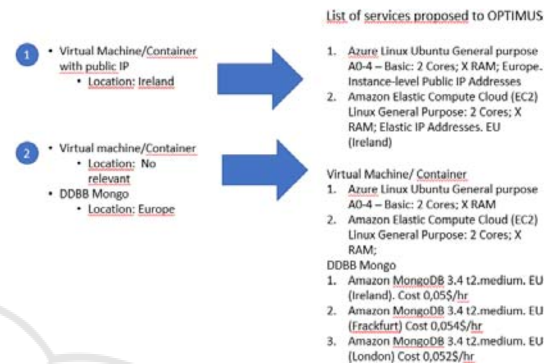

Figure 8: ACSmI Discovery Cloud Services options.

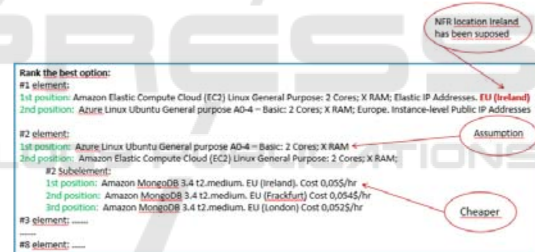

Figure 9: Ranked options for the deployment.

OPTIMUS Simulation process is a complex algorithm allowing the developer to obtain the best deployment schema, based on a set of characteristics that are important for the proper functioning of the multi-cloud application. Taking into account all those inputs and the knowledge about the Cloud Services and how the different combinations of them can impact to the general performance, OPTIMUS Simulation establishes a rank of the Cloud Services in order to assign the first of them to the best Deployment Schema. Considering the Location values and the cost for some of the options, the result of the simulation will be the one shown in the Figure 9.

The developer will accept the Deployment offered and confirm it as the Best Deployment Schema, using the OPTIMUS User Interface.

This deployment schema would be transformed into the concrete deployment script and will be usde to support the Ops (Operation) phase of the multi-cloud application.

5 CONCLUSIONS AND FUTURE WORK

This paper has presented a tool-based approach for the adaptation of the DevOps philosophy to the specific case and needs of applications distributed over different cloud resources (multi-cloud applications).

The solution described intends to solve some of the challenges of multi-cloud applications design. These challenges include:

- Applications need to be responsive to hybrid/multi-cloud model scenario.
- Means shall be provided to manage and assess cloud deployment alternatives to better support cloud resources selection and discovery.
- Existing cloud services shall be made available dynamically, broadly and cross border.

The novel concept and first versions of the tools created for the implementation of the extended *"Dev"* concept has been validated in a sandbox scenario with the SockShop application. Future work will include the complete development (including all the functionalities) of the presented tools (ARCHITECT, OPTIMUS and ACSmI), their integration into a holistic DevOps toolchain and their validations on real business scenarios.

ACKNOWLEDGEMENTS

The project leading to this paper has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731533.

REFERENCES

ARTIST Consortium, 2014. *D9.4 Collection of optimization patterns,* s.l.: s.n.

Bourgon, P., 2017. *Go Kit.* [Online] Available at: https://gokit.io/ [Accessed 10 March 2018].

Chen, L., 2015. Towards Architecting for Continuous Delivery. *In 12th Working IEEE/IFIP Conference on Software Architecture (WICSA).*

DECIDE Consortium, 2017. *DECIDE Grant Agreement,* s.l.: s.n.

Di Nitto, E., Matthews, P., Petcu, D. & Solberg, A., 2017. *Model-Driven Development and Operation of Multi-Cloud Applications.* 1 ed. s.l.:Springer International Publishing.

Docker, n.d. *Docker.* [Online] Available at: https://www.docker.com/ [Accessed November 2017].

Eclipse Cloud Development, 2018. *Eclipse Cloud Development.* [Online] Available at: http://www.eclipse.org/ecd/ [Accessed March 2018].

Edwards, D., 2015. *DevOps paradox: Going Faster Brings Higher Quality, Lower Costs, & Better Outcomes,* s.l.: s.n.

Escalante, M. et al., 2017. *D5.1 ACSmI requirements and technical design,* s.l.: s.n.

Escalante, M. et al., 2017. *D5.2 Initial Advanced Cloud Service meta-Intermediator (ACSmI),* s.l.: s.n.

Farid, L. et al., 2017. *Initial architectural patterns for implementation, deployment and optimisation,* s.l.: s.n.

Fehling, C. et al., 2014. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications.* s.l.:Springer.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J., 1995. *Design Patterns: Elements of Reusable Object-Oriented Software..* s.l.:Addison-Wesley.

Gavilanes, J. et al., 2017. *D2.1-Detailed Requirements Specification,* s.l.: s.n.

IBM BlueMix, 2016. *IBM BlueMix DevOps services.* [Online] Available at: hub.jazz.net [Accessed 2018].

JHipster, 2015. *JHipster.* [Online] Available at: http://www.jhipster.tech/ [Accessed November 2017].

Kopanelli, A. et al., 2015. *A Model Driven Approach for Supporting the Cloud Target Selection Process.* s.l., Elsevier.

Mojtaba, S., 2015. Architecting for DevOps and Continuous Deployment. *In proceedings of the ASWEC 2015 24th Australasian Software Engineering Conference (ASWRC '15 Vol. II).*

Node.js Foundation, n.d. *Node.js.* [Online] Available at: nodejs.org [Accessed 2018 march 2018].

Riungu-Kalliosaari, L. et al., 2016. *DevOps Adoption Benefits and Challenges in Practice: A Case Study.* s.l., International Conference on Product-Focused Software Process Improvement.

Spring, n.d. *Spring Boot.* [Online] Available at: https://projects.spring.io/spring-boot/ [Accessed November 2017].

Steinborn, T., 2018. *The future of DevOps is mastery of multi-cloud environments,* s.l.: Open Source.

UpGuard, 2016. *The four prerequisites for DevOps success,* s.l.: s.n.

Weaveworks, 2016. *SockShop App.* [Online] Available at: https://microservices-demo.github.io/