EasyChair Preprint

№ 548

DevOps' Shift-Left in Practice: An Industrial Case of Application

Miguel Jiménez, Luis F. Rivera, Norha M. Villegas,
Gabriel Tamura, Hausi Müller and Pilar Gallego

October 1, 2018

DevOps' Shift-Left in Practice: An Industrial Case of Application

Miguel Jiménez[1], Luis F. Rivera[2], Norha M. Villegas[2], Gabriel Tamura[2], Hausi Müller[1], and Pilar Gallego[3]

[1] University of Victoria,
Victoria, British Columbia, Canada
{miguel,hausi}@uvic.ca
[2] Universidad Icesi,
Cali, Valle del Cauca, Colombia
{lfrivera,nvillega,gtamura}@icesi.edu.co
[3] Carvajal Organization,
Cali, Valle del Cauca, Colombia
pilar.gallego@carvajal.com

Abstract. DevOps aims at unifying software development and operations to improve products and deliver value to customers. However, many organizations adopt DevOps mainly from a traditional perspective, that is, going forward from development to operations. In this paper we present a case of study that illustrates how Carvajal Technology and Services, a software development organization, improved the design of a family of its software products by exploiting operations data. This case of application constitutes a first incursion of the organization into DevOps, exemplifying how the community and companies in industry can also go backwards from operations to development and design, thus realizing the DevOps *shift-left* concept. The main contributions of this paper are: (i) the analysis of the industrial DevOps application, for which the deployment automation mechanism is crucial to realize the shift-left concept effectively; and (ii) Amelia, the DSL we developed for deploying the different (re)designs to put into operation and gather feedback data rapidly. To evaluate the approach, the organization analyzed this incursion in both directions: from development to operations, on the benefits of deployment automation; and from operations back to development, by improving the throughput of the original design by a factor of five.

Keywords: DevOps, Shift-left in DevOps, Software Deployment, Deployment Automation

1 Introduction

In recent years, the need for delivering added value to end-users as soon and as frequently as possible, even due to small changes, has increased the adoption of DevOps and continuous delivery processes [1]. Several frameworks and tools have been proposed to address this urgency of faster and more frequent software

releases. In general, these frameworks recognize the deployment as an independent phase in the software development life cycle. Given its crucial importance for achieving continuous software delivery, there is a need for a better understanding of the various deployment uses and roles it can adopt for realizing the versatile DevOps principles in both, the forward and backward directions.

On the one hand, many organizations adopt DevOps from a traditional point of view, that is, focusing on deployment as a checkpoint going forward from development to operations, considering it as an end for that purpose. On the other hand, from a wider DevOps perspective, automated deployment is a crucial phase for instance to explore, in the operations setting, different design implementations, enabling the collection of data efficiently. This data, used backwards, is key to improve development, and in this process, deployment serves as a medium rather than as an end. Achieving DevOps requires to find ways of traversing development and operations processes in both directions, and the shift-left concept enforces especially its backward application.

The goal of this paper is to present an industrial application that uses automated deployment as a fundamental mechanism to enable the organization to systematically conduct experiments to collect data from the software operation and improve the design of its reference architecture. This is an example of how organizations can realize the backward application of DevOps, centered on automating the deployment phase. We developed this case in the context of an industrial-academic partnership between Carvajal Technology and Services, and Icesi University. Carvajal is a multinational organization with IT and software development as one of its business units and over 1,200 software developers.

Concretely, for the experiments conducted, we employed and combined different design patterns to produce several architectural configurations. However, the number of configurations to be deployed and executed raised several challenges, such as the repeated deployment and re-deployment of the resulting configurations instantiations, which requires its automation to enable the efficient gathering of performance measurements. We addressed these challenges following the DevOps principles. Solving them required the development of Amelia, a domain-specific language (DSL) for automating deployments. The analysis and evaluation results confirmed the critical role of Amelia for achieving the dual-direction of the DevOps application, and especially of the shift-left concept.

This paper is structured as follows. Section 2 presents the background and relevant DevOps concepts for this work. Section 3 presents Amelia, our language for deployment automation including its main features. Section 4 presents the methodology used to apply and evaluate Amelia in a dual deployment strategy of use to address the industry case of application, and discusses the evaluation results. Finally, Section 5 concludes the paper.

2 Background

This section introduces DevOps principles and relevant concepts of continuous delivery upon which the subject application is built, and where automated deployment is crucial.

2.1 DevOps Adoption Paths and Principles

IBM proposed four paths and respective focus of concern for adopting DevOps: (i) *steer*, focused on continuous business planning; (ii) *develop/test*, on continuous integration and testing; (iii) *deploy*, on continuous release and deployment; and (iv) *operate*, on continuous monitoring [2]. We followed the *deploy* adoption path, on which most of the DevOps inherent concepts and capabilities were originated, including the definition of the delivery pipeline. This pipeline enforces continuous deployment of software to quality assurance and then to production, efficiently and in an automated way.

Similarly, IBM consolidated the main principles developed in the evolution of the DevOps movement [2]: (i) develop and test against production-like systems, the main premise of the shift-left concept moving operations earlier toward development; (ii) deploy with repeatable and reliable processes, for which automation is essential; (iii) monitor and validate operational quality, based on functional and non-functional software characteristics; (iv) amplify feedback loops, reacting and producing changes more rapidly. For the development of this case, we followed these principles in the context of the deploy adoption path.

2.2 Continuous Delivery

Continuous delivery is a software engineering approach—aligned with the Devops principles and the *deploy* adoption path—that promotes to deliver added value to end-users as soon and as frequently as possible, by deploying successful releases of a subject software system [3]. The major benefits of this approach are the empowerment of teamwork between development and operations, the injection of fewer bugs (therefore reducing costs and risks), generation of less pre-release team stress, and a more flexible deployment process. To achieve these benefits, a software provider must promote a culture of collaboration between all teams involved in the delivery process, the sharing of knowledge and tools among participants, the establishment of measurement metrics, and the gathering of regular feedback for continuous improvement. That is, software providers must subscribe to DevOps principles to acquire continuous delivery benefits [4] and guarantee a repeatable and reliable process for releasing software, the automation of deployment and operation activities, the automation of integration, testing, and release processes, and the definition of an effective quality assurance process [3].

2.3 The Deployment Life Cycle

Deployment has been characterized by the Object Management Group (OMG) and others as "the process composed of interrelated and evolving activities that comprise the lifecycle of a particular system to be brought into—and out of—service" [5,6,7,8], as depicted in Figure 1.

The main activities of the deployment lifecycle are described as follows: *release* is the bridge between development and deployment. It comprises all necessary tasks to prepare, package, and provide (*e.g.*, via its publication) a software

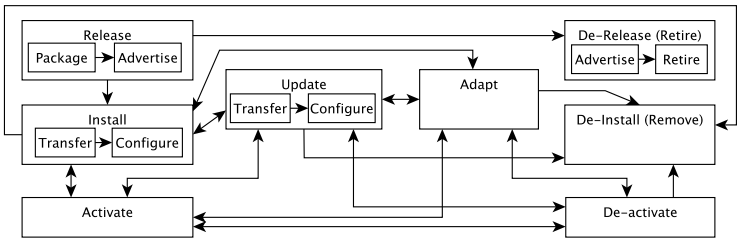

Fig. 1: The deployment process adapted from [6].

product for deployment into consumer sites. The released package must contain sufficient metadata to describe the resources by which the software depends on to correctly execute in the target environment, such as libraries, configuration files and executables. *Installation* encompasses all configuration operations and assembly of the resources to prepare the software system for activation. Installation involves the transfer of the software components from the producer site to the consumer sites. *Activation* allows the consumer to actually use the software. In the case of simple software, the activation is usually realized through the creation of a command for executing a binary component of the application. The activation of complex software involves several components that must be executed in a particular sequence. *Deactivation* means to stop any running component of the software system. *Update* is a special case of installation triggered by the release of a new version of the product or of any of its components. However, it may require the deactivation of the software (or the component to be updated) before executing any operation. *Adaptation* is similar to the update activity, in the sense that both modify a previously installed software system. However, an adaptation is triggered by context changes with the goal of assuring the accomplishment of properties or requirements in the deployed system. An adaptation may be performed autonomously in the form of self-adaptation while the subject system is running. *Deinstallation* is the activity performed when the deployed software system is no longer required at the consumer site. *Derelease* (retirement) is the process of finishing the support for a software system or a given configuration of a software system. Retiring a system makes it unavailable for future deployments.

3 Shifting Operations Left by Automating Deployment

This section presents our subject application and addressed challenges, and illustrates how we realized the shift-left conceptual movement of operations toward development. For this realization, the key enabler for exploiting operations results back into design, efficiently, was the automation of the deployment phase. We performed this automation with Amelia, a domain-specific language (DSL) we developed for this purpose.

3.1 The Industrial Case of Application

The case of application mainly answers the question of how to satisfy the performance requirements of the core engine of a software product family that processes large XML files for different application domains, established as a product's quality attribute by a set of corporate clients. Of course, this is a problem of design, critical for a reference architecture that affects a entire family of products. Nonetheless, despite a design problem, its effective solution requires to move operations earlier toward development.

Similar to other cases of development in the organization, the initial software design for processing XML documents had basically a 3-tier architecture, relying on the computing power of the assumed infrastructure. However, a not so early test on this infrastructure showed serious performance limitations. At this point, the organization decided to look for alternatives, and our research group at Icesi University joined Carvajal for its first incursion into DevOps.

As an exemplar of the product family, we selected a concrete product that Carvajal developed for the Colombian National Agency for Overcoming Extreme Poverty (ANSPE—initials corresponding to the Spanish name[4]) to allow census workers to collect demographic data in mobile devices offline (*i.e.*, in regions with no access to telecommunications). After days or weeks, hundred of workers synchronize the collected census data with a centralized server, from around the whole country. This synchronization suffered from severe delays and timeout errors, due to the large number of requests overloading the central server.

3.2 Addressed Challenges

Processing XML files is a common, CPU-intensive task that supports core business processes in different domains, ranging from plain data transmission and transformation to full data interoperability, for all of which there exist several libraries and processing strategies. However, it is non trivial which strategy and libraries to select in a solution design space at large scale, whose primary requirement is on performance, given the combined implications they have on this quality attribute. Moreover, these decisions must be considered in distributed processing scenarios, such as the one illustrated in Figure 2. As a result, most of the challenges we addressed were referred to as deployment issues, as follows:

Variability in architectural configurations and instantiations. Adding to the number of XML processing strategies and libraries, there are also a number of domain-specific design patterns to consider among those for improving performance, such as Producer/Consumer, Master/Worker, Reactor, among others. These design patterns' components, along with the application's software components can be deployed in different processing nodes (cf. Fig. 2 for example), yielding several architectural configurations and variations. That is, for each architectural configuration, several instantiations are possible (*e.g.*, varying the number of slave processing nodes). Each of these instantiations implies

[4] Agencia Nacional para la Superación de la Pobreza Extrema

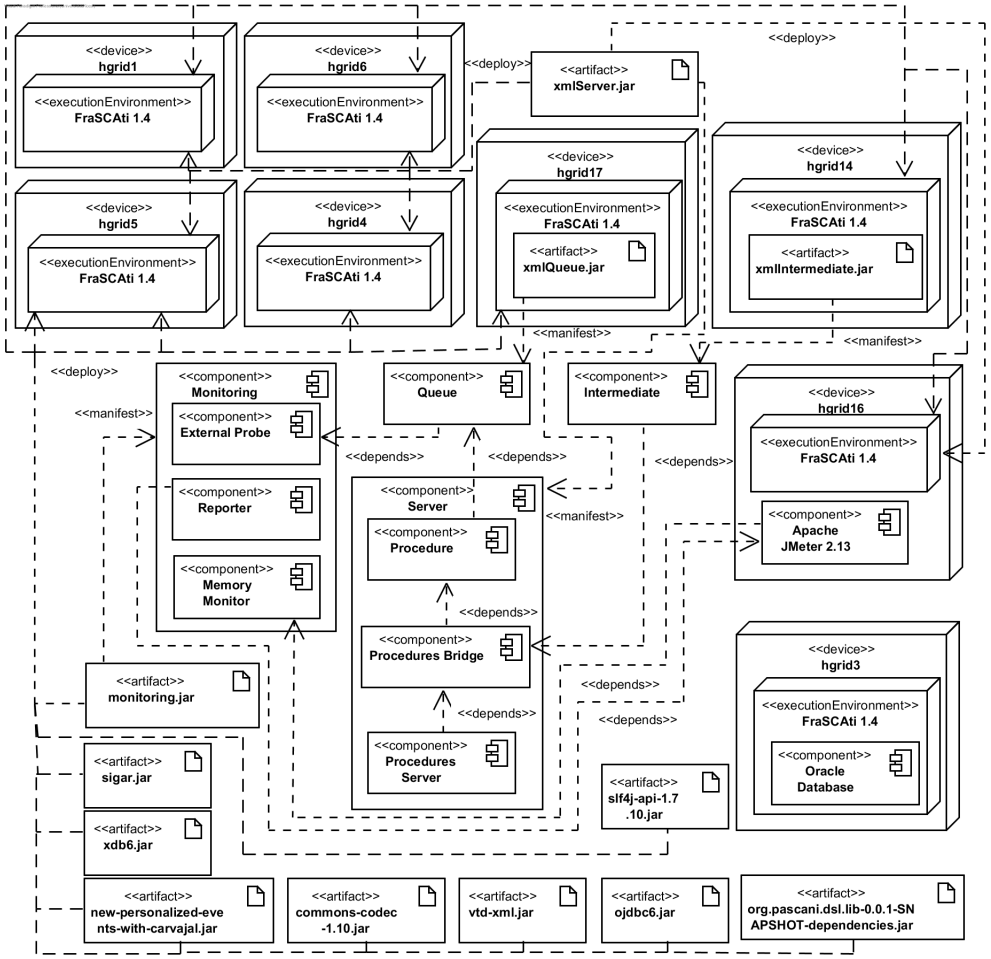

Fig. 2: Possible Deployment Diagram for the ANSPE.

a corresponding deployment and execution process to test on operations, whose variations are intricate and their combinations large in number.

Dependency management. Dependencies exist in all of the deployment variations, and along all the deployment process phases, both among software components, and among software and hardware components.

Coordination control. The execution of the deployment tasks and phases, not only within a particular processing node but also among all of them, must be coordinated and controlled along all consumer sites, especially in the case of distributed software systems such as the ANSPE. This coordination and control must observe the deployment dependencies of each particular application.

Modularity and composability. Deployment tasks and phases should be specifiable in independent but composable modules. Specification encapsulation and modularity should allow abstraction scalability and factoring out common and

repeatable deployment tasks in separate modules, each having well defined interfaces and internal specifications, and enabling the integration of other specifications. However, composability increases the complexity of dependency management through the entire deployment life cycle.

Reusability. Deployment specifications should be reusable in different deployment workflows. These specifications could be provided as part of a catalogue to facilitate deployment design and specification.Reusability also increases the complexity of dependency management.

Extensibility. Refers to the functional ability to easily override or extend the behaviour of an existing deployment specification. Without adequate extensibility capabilities, a deployment specification must be either modified in place or duplicated to adapt it to be used in a different deployment context, which reduces reusability.

3.3 The Amelia DSL for Automating Deployment

Automatically deploying, configuring and executing scalable software-intensive systems, whose components are distributed in several processing nodes and feature runtime dependencies among them, is not a trivial task. For instance, even in systems such as Docker [9], the way to resolve runtime dependencies in a distributed software deployment and execution scenario is left to the developer—checking for them in the application's subsystems source code, which is not appropriate nor possible in all scenarios. Therefore, we adopt the following lean strategy for automating deployment. Instead of using heavy-weight and multi-function systems like Kubernetes [10] and Apache Mesos [11], which were originally invented for cluster management (*i.e.*, not exactly focused on deployment automation), we designed and developed Amelia, a compact DSL tailored for specifying and executing deployment workflows for distributed software systems. Amelia contains both declarative and imperative statements that facilitate coordinated control of the overall deployment process, and at the same time, offer granular control over all of the executed operations. Amelia is based on the Xbase expression language [12] and is fully integrated with the Java type system. This integration allows not only to reuse existing Java code, but also to extend the Java's base library.

An Amelia file specification can be written as either a subsystem description or a deployment strategy. The first one is a modular unit representing the overall structure of the (sub)system to deploy and corresponding deployment operations; a subsystem description is composed of, and dependent on, other subsystems, thus supporting modularity, dependency management, composability and reusability. The latter is an execution flow specification that dictates how to perform the deployment operations, thus supporting coordination and control. For example, Amelia allows to retry tasks upon encountering a failure, or systematically repeat the same deployment procedure, which is useful for instance to "warm up" a system before running performance tests. The current implementation of Amelia can be used as a standalone compiler or as an Eclipse plug-in. From a specification file, the compiler generates an executable

Java application that automatically resolves the subsystem dependencies and (sub)module inclusions while logging the result of deployment task execution.

Language Structure and Concepts. Listing 1.1 presents an example of a system description for the deployment of the ANSPE `Monitor` subsystem.[5] An AMELIA specification is comprised of a package declaration section, an optional Java import section, an optional extension section, and a type declaration section (*cf.*, respective highlighted regions ❶, ❷, ❸, ❹, in the listing). The characteristics of each section are presented as follows.

Subsystems. A `subsystem` collects and encapsulates the sets of variables, parameters, and execution rules for a software subsystem to be deployed. Local variables within a subsystem may control the execution flow (*e.g.*, variable `compileMonitor` in Listing 1.1). Parameters are used to configure a subsystem according to a deployment strategy. They are included in the subsystem's constructor in the same order they are defined. In Listing 1.1, the (implicit) constructor of the `Monitor` subsystem includes the given name for the monitor, the communication protocol, and the target architecture. Execution rules group commands that describe the deployment lifecycle of a subsystem; that is, they represent the various phases of a subsystem's deployment. The deployment of the `Monitor` subsystem is described as follows: *initialization* checks whether the monitor's source code must be compiled; *compilation* generates an executable artifact by compiling the monitor's source code; and *activation* executes such artefact passing the corresponding arguments.

There is no main entry point or main function in a subsystem specification where the deployment execution starts. Instead, the execution flow is expressed and controlled by rules and their dependencies. Thus, any rule not depending on any other rule is triggered immediately upon deployment execution. For example, in Listing 1.1, the `compilation` rule depends on the local rule `initialization`; that means that the first command from the former will be executed after the last command from the latter. A group of commands within an execution rule is executed sequentially. A rule (*i.e.*, its commands) is executed on at least one host and may depend on a Boolean expression (*e.g.*, line 44 in Listing 1.1). Conditional expressions may be placed next to the host(s) to guard the execution of a set of rules on a host. In case the Boolean expression is false, the set of rules is not executed and the dependent rules are released from the dependency.

AMELIA has built-in support for five commands: transfer files, change the working directory, compile FRASCATI [13] components, run FRASCATI components, and execute other commands (*e.g.*, line 41 in Listing 1.1).

The extension section of a subsystem allows expressing subsystem dependencies and inclusions. In the former case, the execution of all subsystem's rules depends on the successful execution of the rules defined within the subsystem dependencies. In the latter case, both parameters and execution rules from the

[5] The complete versions of all examples discussed in this paper are available in the AMELIA evaluation repository `https://github.com/unicesi/amelia-evaluation`

```
1  package co.edu.icesi.driso.amelia.carvajal.xml.large
2
3  import java.util.concurrent.atomic.AtomicBoolean
4  import co.edu.icesi.driso.amelia.carvajal.xml.large.CommProtocol
5  import static extension examples.Util.warn
6
7  includes co.edu.icesi.driso.amelia.carvajal.xml.large.CommonSpecification
8  depends on co.edu.icesi.driso.amelia.carvajal.xml.large.ApacheServer
9
10 // A Subsystem for the monitoring component.
11 subsystem Monitor  {
12   // The name of the monitor
13   param String name
14
15   // The protocol used for communication among components
16   param CommProtocol protocol
17
18   // The target architecture
19   param Arch arch
20
21   // Activation variables.
22   var String component = "memory-monitor"
23   var String service = "startMemoryMonitor"
24   var String method = "startMemoryMonitoring"
25
26   // Whether to compile the monitor
27   var AtomicBoolean compileMonitor = new AtomicBoolean
28
29   // The classpath used for compiling the monitoring component
30   var String[] _classpath = #[
31     'org.pascani.dsl.lib-0.0.1-SNAPSHOT-dependencies.jar',
32     'new-personalized-events-with-carvajal.jar',
33     'slf4j-api-1.7.10.jar',
34     'sigar.jar']
35
36   // The libpath used for activating the monitoring component
37   var String _libpath = classpath + #[component]
38
39   on CommonSpecification.host {
40     initialization: CommonSpecification.changeDirectory;
41       (cmd 'ls')
42         .fetch[compileMonitor.set(!it.contains('«component».jar'))]
43
44     compilation ? compileMonitor: initialization;
45       compile CommonSpecification.actionFolder component -classpath _classpath
46
47     activation ? protocol == CommProtocol.RMI ||
48                  protocol == CommProtocol.REST: compilation;
49       (run component -libpath _libpath -s service -m method -p #[name, arch])
50         .warn('Executing the Monitor, could take several minutes...')
51   }
52 }
```

Listing 1.1: A subsystem specification to deploy a monitor in ANSPE

```
1  package co.edu.icesi.driso.amelia.carvajal.xml.large
2
3  import java.util.Map
4  import org.amelia.dsl.lib.descriptors.Host
5  import org.amelia.dsl.lib.util.Hosts
6  import org.amelia.dsl.lib.util.RetryableDeployment
7  import examples.ubuntu.Arch
8  import co.edu.icesi.driso.amelia.carvajal.xml.large.CommProtocol
9
10 includes co.edu.icesi.driso.amelia.carvajal.xml.large.Monitor
11
12 // Deployment of a monitor component.
13 deployment DeploymentMonitor {
14   // Load all hosts and then filter
15   val Map<String, Host> hs = Hosts.hosts("hosts.txt").toMap[h|h.identifier]
16   val executionHosts = #[hs.get("hgrid1"), hs.get("hgrid4"), hs.get("hgrid5")]
17   for(i: 0..executionHosts.size - 1) {
18     add(new Monitor("monitor/src", 'mntr«i»', CommProtocol.RMI, Arch.amd64))
19   }
20   val helper = new RetryableDeployment()
21   helper.deploy([ start(true) ], 2) // Deploy everything & retry if it fails
22 }
```

Listing 1.2: A deployment strategy for the ANSPE Monitor subsystem

included subsystems are made part of the including subsystem. This means that its constructor is modified to accept the included parameters, and that included rules are dependent on local rules. In Listing 1.1, `CommonSpecification.host` refers to a parameter defined in subsystem `CommonSpecification`. In the same way, the local rule `initialization` depends on the `changeDirectory` rule defined in subsystem `CommonSpecification`. In this case, the dependency uses the rule's qualified name to avoid a name collision with a local rule.

Deployment Strategies. Deployment strategies are simpler than subsystems. Their purpose is twofold: i) to configure subsystem instances; and ii) to determine how many times, and including which subsystems, must the system be deployed.

Listing 1.2 presents the monitoring deployment strategy in detail. Line 18 shows how to instantiate the subsystem `Monitor`. To make a subsystem available to deploy, the subsystem must be included in the extension section.

An invocation to the static method `start` initiates the deployment execution. Notice that it can be invoked more than once, always blocking until the deployment finishes. Line 21 features a utility class that invokes a lambda function as many times as specified if its execution throws an error. In this case, if the deployment fails this helper will restart it once.

Extensions. AMELIA supports two Java-based extension mechanisms. The first one allows creating new commands by instantiating a `CommandDescriptor`—a Java class defined in the Java runtime library. The second mechanism allows augmenting a command by extending its behaviour. For example, `fetch` (*cf.* line

42 in Listing 1.1) is an extension that implicitly import a static method that return a new command; that is, a `CommandDescriptor` instance wrapping another. The extension retrieves the output from the input command and passes it to a lambda function. Extensions could be chained together.

3.4 Automated Deployment Execution

Performing the deployment of the various architecture configurations and variants is a demanding task, even with automated tools. It requires specifying the deployment of each architectural configuration, whose variants and instantiations requires repeating same portions of the specifications. Amelia addresses this problem allowing parameterised and reusable modules and procedures to specify different configuration instantiations. Each specification comprises the deployment and configuration operations required to deploy and execute each of the software components. Of course, some of these specifications are reusable to the extent to which the deployment language or mechanism supports modularity and encapsulation. For example, the Master/Worker design pattern could be deployed by creating two types of specification, a master and a worker component, whose computing node is a parameter. Encapsulation is critical to avoid side effects. In Amelia, these specifications are subsystems, which define explicit instantiation parameters and a clear interface based on the deployment operations. Moreover, the automatic execution of these variations is demanding as well. For instance, the associated performance measurements need to be statistically significant. Therefore, it is not enough to gather data from a single execution. Deployment strategies in Amelia facilitate this task by allowing the deployer to setup the subsystems and their parameters per each execution, and to repeat it as necessary.

Furthermore, systematic deployment execution can be further exploited in Amelia. Each execution cycle may comprise from configuration and deployment to measurement gathering and metrics storing. In fact, new deployments can be stopped when a previous instance already presents better metrics. Amelia's deployment strategies allow this by providing granular control over the deployment start and end, tight integration with the Java programming language, and control over first-class deployment concepts, such as subsystems and their parameters.

4 Evaluation

This section presents the analysis and evaluation of the case of application, aimed at assessing how to exploit automated deployment in the DevOps process chain, not only in the forward (*i.e.*, traditional), but also in the backward direction.

4.1 Qualitative Analysis

In the forward direction, going from development to operations, we performed a qualitative evaluation of the language effectiveness for specifying the deployment of different architectural configurations of the ANSPE software system. For this evaluation, we designed an evaluation protocol that was applied by

six master students, some of them members of the Carvajal engineering staff. They developed several AMELIA specifications from UML deployment diagrams representing some ANSPE architecture configuration variants. These were variants of two design patterns: Producer/Consumer and Reactor; all variants used RMI as the communication protocol and executed on the same middleware. The first configuration required four processing nodes, while the second, twelve; in both cases, only one Consumer component is deployed on each processing node. Then, the participants completed a questionnaire regarding their experience. Based on their answers, we evaluated the language effectiveness using FQAD, a Framework for the Qualitative Assessment of DSLs [14], which refines a subset of quality characteristics defined in the ISO/IEC 25010:2011 standard. Carvajal participated in the selection of the quality characteristics to evaluate: functional suitability, usability, reliability, productivity, and expressiveness, as defined in the standard.

In summary, AMELIA as a DSL for deployment automation was evaluated having either "full support" or "strong support" in all of these characteristics.[6] Among these, functional suitability, defined as the degree in which the DSL supports completely and appropriately the specification of scripts to automate the deployment of distributed software, along with productivity and reliability, are of major importance for evidencing deployment automation benefits for the organization.

4.2 Quantitative Analysis

In this section, we present the quantitative evaluation of AMELIA as the vehicle that enables the shift-left realization (*i.e.*, the backward direction of the DevOps application). For this evaluation, we took advantage of the qualitative evaluation results, that is, by directing the master students to refine and complete the deployment scripts specified for that evaluation, and executing them to finally obtain a design that satisfies the performance requirements.

The quantitative evaluation follows an experiment design approach, in four steps. First, we started by selecting a set of domain-specific design patterns for improving performance, suitable for our case. Second, we defined several architectural configurations, variations and corresponding instantiations that result from the incorporation of the selected design patterns to the ANSPE's reference architecture, and further adapted and completed correspondingly the AMELIA deployment scripts. Third, we prepared a hardware infrastructure as close as possible to the organization's one, with controlled conditions, and executed the experiments by running the deployment scripts for each of the architectural instantiations. From these executions, the metrics of system performance were collected. Finally, we compared the performance metrics to select the best architectural configuration.

In the following sections, we explain the experiment design, its execution and the analysis of results.

[6] Given the space restrictions, we omit in this paper the details of the evaluation protocol and its execution. Nonetheless, the evaluation files are available in the AMELIA evaluation github repository `https://github.com/unicesi/amelia-evaluation`

Experiment Design. The experiment design involved the generation of combinatorial instantiations of architectural configurations and factors, where the objective was to compare the performance of these configurations instatiations, measured systematically. The experimental unit encompassed the different system architectural configurations. The response variable under study was the performance (throughput) in number of XML files processed per minute. The controllable input factors included (i) the set of XML files with CRUD operations to update the census and demographic status of families; (ii) the design patterns relevant for distributed processing (*e.g.*, Producer/Consumer (P/C) and Reactor); (iii) the communications protocol (provided by a middleware); and (iv) the number of working components that were deployed on each slave processing node (*i.e.*, consumers in P/C, controllers in Reactor). The uncontrollable input factors comprised the clock synchronization issues among processing nodes[7], and indispensable operating system processes. Table 1 depicts a summarized version of the experiment design[8], for XML files of 1Mb of application operations, and 12 processing nodes (*i.e.*, computers with 1 quadcore CPU).

Design Pattern	**Communications Protocol**	**No. Consumers/ Controllers**
P/C	RMI	12
P/C	RMI	48
P/C	RMI	96
P/C	Ice	12
P/C	Ice	96
Reactor	RMI	12
Reactor	Ice	12

Table 1: Factors instantiated for the experiment design.

Experiments Execution. The experiments were executed in an infrastructure equipped with 22 computers configured with one CPU Intel i7 quadcore, and 16 Gb of RAM running Linux Fedora 25, although not all computers were used at the same time in any of the experiments. As part of the configuration process in the deployment specification, all useless and non-critical operating system services were stopped before executing the experiments.

The steps for performing the deployment of the architectural configuration instantiations included an important point in the pre-requisites of execution: the database required to be restored with data backed-up exactly in the date previous to those of the XML files to be synchronized. This implied that, previous to executing any experiment, restoring the database was necessary.

The deployment and execution of each of the configurations of the experiment design was repeated at least three times, and the performance metrics averaged.

[7] We used the Precision Time Protocol (PTP, IEEE 1588).

[8] The complete version of the experiment design comprises a set of 324 configurations resulting from the following number of instantiations: 4 patterns x 3 numbers of consumers/controllers x 3 numbers of processing nodes x 3 communication protocols x 3 sizes of input files. We present a subset because of space restrictions.

This means that a total of 21 deployments and executions were performed for the subset of the experiments indicated in Table 1. The role played by Amelia for executing these experiments was really critical.[9]

Analysis of Results. The reference throughput baseline given was of 1 XML file every 32 seconds (that is, 1.87 XML files per minute). Thus, we present the quantitative results of the experiments in Table 2 in this form.

Experiment Configuration	Throughput
P/C, RMI, 12 components	1 file / 11,35 sec
P/C, RMI, 48 components	1 file / 6,9 sec
P/C, RMI, 96 components	1 file / 6,7 sec
P/C, Ice, 12 components	1 file / 11,51 sec
P/C, Ice, 96 components	1 file / 6,1 sec
Reactor, RMI, 12 components	1 file / 21,95 sec
Reactor, Ice, 12 components	1 file / 20,51 sec

Table 2: Execution results of the experiment.

According to these results, the best architectural configuration uses the Producer/Consumer design pattern with Ice as communication protocol, distributing 96 consumers in 12 processing nodes, for a throughput of 1 file every 6,1 seconds, around five times better than the reference baseline.[10]

In addition, it is worth noting two other points: first, that Producer/Consumer is consistently and significantly better than Reactor, and second, that even though Ice is in general better than RMI, the difference is not that significant.

5 Conclusions

Conveying good design practices into actual architectural configurations with the purpose of guaranteeing specific quality metrics is better performed when having evidence gathered from experimentation with systems in operation. Automated deployment is crucial for performing this experimentation for an architect to select, for instance, the most appropriate design pattern. Thus, we believe that, in the context of DevOps, realizing automated deployment as presented in this paper, is important to make informed decisions in the development process based on factual data gathered from operations.

In this paper we have reported on an industrial case of application regarding the dual use of automated deployment in the both directions of a DevOps setting: from development to operations and also in the inverse direction, effectively realizing the shift-left concept. To this end, we illustrated how our Amelia DSL facilitates the automation of the deployment process for various architecture variants and configurations, following the DevOps principles and the *deploy* adoption path.

[9] In the complete experiments set, a total of 972 deployments and respective executions were performed.

[10] In fact, this configuration was the best among the 324 in the complete experiment.

We applied FQAD to evaluate five quality characteristics in Amelia, confirming its effectiveness for deployment automation. For the quantitative evaluation, we used Amelia to specify and deploy 324 architectural configurations and variations in a set of experiments. We used the experiments results to redesign the reference architecture of the system presented in the case, improving its throughput by a factor of five. One of our findings in this case is that the Producer/Consumer design pattern is consistently better than Reactor for improving throughput.

Acknowledgments. This work was funded in part by the National Sciences and Engineering Research Council (NSERC) of Canada, IBM Canada Ltd. and IBM Advanced Studies (CAS), the University of Victoria, Universidad Icesi (Colombia), and Organización Carvajal SA (Colombia).

References

1. Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. 1st edn. Addison-Wesley Professional
2. Sharma, S., Coyne, B.: DevOps for Dummies. 3rd Limited IBM edn. John Wiley & Sons (2017)
3. Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. 1st edn. Addison-Wesley Professional (2010)
4. Humble, J., Molesky, J.: Why enterprises must adopt devops to enable continuous delivery. Cutter IT Journal **24**(8) (2011) 6–12 cited By 32.
5. OMG, D.: Configuration of component-based distributed applications specification—version 4.0
6. Carzaniga, A., Fuggetta, A., Hall, R.S., Heimbigner, D., Van Der Hoek, A., Wolf, A.L.: A characterization framework for software deployment technologies. Technical report, DTIC Document
7. Hall, R.S., Heimbigner, D., Wolf, A.L.: A cooperative approach to support software deployment using the software dock. In: Proceedings of the 21st International Conference on Software Engineering. ICSE '99, ACM 174–183
8. Dearle, A.: Software deployment, past, present and future. In: 2007 Future of Software Engineering. FOSE '07, IEEE Computer Society 269–284
9. Willis, J.: Docker and the three ways of devops. Technical report, Docker Inc.
10. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J.: Borg, omega, and kubernetes. **59**(5) 50–57
11. Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A.D., Katz, R.H., Shenker, S., Stoica, I.: Mesos: A platform for fine-grained resource sharing in the data center. In: NSDI. Volume 11. 22–22
12. Efftinge, S., Eysholdt, M., Köhnlein, J., Zarnekow, S., von Massow, R., Hasselbring, W., Hanus, M.: Xbase: Implementing domain-specific languages for java. **48**(3) 112–121
13. Seinturier, L., Merle, P., Rouvoy, R., Romero, D., Schiavoni, V., Stefani, J.B.: A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures. **42**(5) 559–583
14. Kahraman, G., Bilgen, S.: A framework for qualitative assessment of domain-specific languages. **14**(4) 1505–1526