From Theory to Practice: The Challenges of a DevOps Infrastructure as Code Implementation

Clauirton Siebra[1], Rosberg Lacerda[2], Italo Cerqueira[2], Jonysberg P. Quintino[2], Fabiana Florentin[3], Fabio Q. B. da Silva[4] and Andre L. M. Santos[4]

[1]*Informatics Center, Federal University of Paraiba, Joao Pessoa-PB, Brazil*
[2]*CIn/Samsung Project, Centro de Informática, Universidade Federal de Pernambuco, Recife-PE, Brazil*
[3]*SIDI/Samsung, Campinas-SP, Brazil*
[4]*Centro de Informática, Universidade Federal de Pernambuco, Recife-PE, Brazil*

Keywords: DevOps, Infrastructure as Code, Automation, Software Delivery Management.

Abstract: DevOps is a recent approach that intends to improve the collaboration between development and IT operations teams, in order to establish a continuous and efficient deployment process. Previous studies show that DevOps is based on dimensions, such as culture of collaboration, automation and monitoring. However, few studies discuss the current frameworks that support such dimensions, so that there is a lack in information that could assist development teams in deciding for the most adequate framework according to their needs. This work aims at presenting a practical DevOps implementation and analysing how the process of software delivery and infrastructure changes was automated. Our approach follows the principles of infrastructure as code, where a configuration platform – PowerShell DSC – was used to automatically define reliable environments for continuous software delivery. Then, we compare this approach with other alternative such as Chef and Puppet tools, stressing the features, advantages and challenges of each strategy. The lessons learned from this work are then used to create a more concrete set of practices that could assist the transition from traditional approaches to an automation process of continuous software delivery.

1 INTRODUCTION

The lifecycle of an application involves teams that usually work in distinct areas and have incompatible goals. For example, while development team wants agility; the operation team is more focused on stability issues. In such domains, applications are manually handed over between these teams with minimal communication. Such separation between entities, which are in fact dependent, translates into an increased time to market and negatively impacts the software quality, decreasing the actual value of the product (Humble and Farley, 2010).

The fundamental conflict in the software process environment is between developers, which have to produce changes at a rapid pace; and IT Operators, which have to maintain infrastructure configuration and availability along these changes. The term DevOps, which is a blend of the Developers and Operations words, is a concept that assists to facilitate these changes (Claps et al., 2015). It builds a living bridge between development and operations and gives them an opportunity to work and collaborate effectively and seamlessly. According to Loukides (2012), *DevOps is a culture, movement or practice that emphasizes the collaboration and communication of both software developers and other information-technology (IT) professionals while automating the process of software delivery and infrastructure changes. It aims at establishing a culture and environment where building, testing, and releasing software, can happen rapidly, frequently, and more reliably*.

Previous works on DevOps (Lwakatare et al., 2015; Hosono, 2012) are mainly focused on propose conceptual frameworks, which intend to create a consensus to the own DevOps definition and their features. Some elements such as culture of collaboration, automation and monitoring; emerged from these works and seem to be the basis for the implementation of DevOps environments. However, while DevOps is becoming very popular between software practitioners; there is still a lack in discussions on frameworks that support its

Siebra, C., Lacerda, R., Cerqueira, I., Quintino, J., Florentin, F., Silva, F. and Santos, A.
From Theory to Practice: The Challenges of a DevOps Infrastructure as Code Implementation.
DOI: 10.5220/0006826104270436
In *Proceedings of the 13th International Conference on Software Technologies (ICSOFT 2018)*, pages 427-436
ISBN: 978-989-758-320-9

implementation and reports of real experiences that could assist development teams in adopting the DevOps principles (Erich et al., 2014; Dyck et al., 2015).

The main focus of this work is on the automation dimension, where the definition of practices related to infrastructure as code creates the basis for an automated process of continuous integration and delivery. Handling infrastructure as code, the following benefits can be obtained (Punjabi and Bajaj, 2016):

- Code can be thoroughly tested to reproduce infrastructure consistently at scale;
- Developers could be provided with a simulated production environment, which increases testability and reliability;
- Infrastructure code can be versioned;
- Infrastructure can be provisioned and configured on demand;
- Proactive recovering from failures can be carried out by continuous monitoring of the environment for violations, which can trigger automatic execution of scripts for rollback or recovery.

Our approach follows the principles of infrastructure as code, where a configuration platform, *PowerShell DSC* (Desired State Configuration) is used to automatically define reliable environments for continuous software delivery. The implementation of this strategy in our organization has generated a set of lessons learned which form the basis for the definition of a more concrete set of practices, which can extend current conceptual models and facilitate the transition from theoretical aspects to pragmatic uses.

The remainder of this paper is structured as follows: Section 2 summarizes the studies on the automation dimension of DevOps, where the focus is on the infrastructure as code aspects and their implementations. Section 3 presents how DevOps concepts were implemented in our organization and the lessons learned from this experience. Section 4 consolidates the lessons learned in our experience in a set of practices that show how to integrate our infrastructure as code strategy to the development process. Furthermore, other approaches for infrastructure as code are analysed and compared with our approach. Finally, Section V concludes this work, stressing the challenges of DevOps implementation and future works that we intend to carry out.

2 STRUCTURE AS CODE

The creation of a DevOps environment is based on principles such as culture of collaboration (Bang et al., 2013; DeGrandis, 2011; Wettinger et al., 2014; Tessem and Iden, 2008; Walls, 2013), measurement of development efforts (Liu et al., 2014; Shang, 2015; Bruneo et al., 2014) and monitoring of system health (Bang et al., 2013; Shang, 2015; Bruneo et al., 2014). However, according to Ebert *et al* (2016), the most important shift over the adoption of DevOps is to treat infrastructure as code, since infrastructure can be shared, tested, and version controlled. Furthermore, development and production could share a homogenous infrastructure, reducing problems and bugs due to different infrastructure configurations. This section discusses the main ideas of this approach and resources that support it.

2.1 Basic Concepts

Infrastructure as Code (IaC) is a DevOps principle used to address problems regarding the manual process of configuration management by means of automatic provision and configuration of infrastructural resources. In this way, the IaC concept is used to describe the idea that almost all actions performed to the infrastructure can be automated. As any code, developers could create automation logic for different tasks such as to deploy, configure and upgrade computational systems and infrastructures. Patterns to use the infrastructure as code were proposed in (Duvall, 2011) and they can be summarized as:

- Automate Provisioning: automate the process of configuring environments to include networks, external services, and infrastructure;
- Behavior-Driven Monitoring: automate tests to verify the behavior of the infrastructure;
- Immune System: deploy software one instance at a time while conducting behavior-driven monitoring. If an error is detected during the incremental deployment, a Rollback Release must be initiated to revert changes;
- Lockdown Environments: lock down shared environments from unauthorized external and internal usage, including operations staff. All changes must be versioned and applied through automation;
- Production-Like Environments: development and production environments must be as similar as possible.

These patterns show that DevOps pushes automation from the development to the infrastructure. Compared with manual infrastructure provisioning, for example, configuration management tools can reduce production provisioning and configuration maintenance complexity while enabling recreation of the production system on the development machines. As discussed in (Ebert et al., 2012), tools are a major DevOps enabler and they are mandatory in automating these and other patterns and tasks. In fact, DevOps considers deliveries with short cycle time. This feature comes from one of the Lean/Agile principles, which stands for "Build incrementally with fast integrated learning cycles". Thus, such strategy requires a high degree of automation, so that it is fundamental the appropriate choice of tools .See a list of tools in (Ebert et al., 2012).

2.2 Tools for Configuration Management

Configuration management tools are the main resources to implement IaC strategies. Such tools aim at replacing error-prone shell scripts, which are employed to manage the state of machines or environments where development codes are going to execute. Shell scripts are potentially complex to maintain and evolve, since they are neither modular nor reusable. Thus, the aim of approaches for configuration management was to provide languages to specify configuration properties without the limitations (low modularity and reusability) of shell scripts. Three examples of these languages, which follow different implementation strategies, are:

- Puppet: domain specific language implemented in a common programming language (originally Ruby, but with newer versions in C++ and Clojure);
- Chef: uses an existing language (Ruby) for writing system configuration "recipes";
- CFEngine: domain specific language also implemented in a common programming language (C).

These languages are often declarative. This means, they describe the desired state of the system rather than a way to achieve it. There are other languages such as Nix, which is a purely functional programming language with specific properties for configuration; and IBM Tivoli System Automation for Multiplatforms. These languages have similar features but may present particular purposes. The IBM approach, for example, facilitates the automatic switching of users, applications and data from one database system to another in a cluster.

Puppet, Chef and CFEngine are the most popular configuration management alternatives. Therefore, it is important to understand some slight differences among them (Younge et al., 2011). Chef and Puppet are very similar since they are based on Ruby. However, Chef seems to present less security vulnerabilities than Puppet. Both languages are more "Ops-friendly" due to its model-driven approach. They also present a relatively small learning curve. Differently, CFEngine is more "Dev-friendly" and its learning curve is steep. However, as advantage, CFEngine has a dramatically smaller memory footprint, runs faster and has far fewer dependencies since it was developed with C. For configuration information, CFEngine uses its own declarative language to create "promises," or policy statements. Puppet, on the other hand, uses a Ruby Domain-Specific Language (DSL) to create its manifests. So those with some Ruby experience may find themselves in more familiar territory with Puppet.

A comparison among these and several other open-source configuration management approaches can be seen in (O'Connor et al., 2017).

2.3 Frameworks

As applications need to be developed and tested in production like environments, some organizations are using strategies such as virtualization and more recently containerization (Scheepers, 2015) to make such environments portable. However, these approaches are also hard to use when they are manually maintained. This scenario motivated the creation of frameworks for setup of more complex development environments.

Two popular examples of frameworks are Vagrant and Docker. Vagrant (Peacock, 2015) is a management and support framework to virtualization of development environments. Instead of running all projects locally on a unique computer, having to rearrange the different requirements and dependencies of each project, this framework allows to run each project in its own dedicated virtual environment. Docker (Miell and Sayers, 2016) is a container-based approach that provides virtualization at the operating system level and uses the host kernel to run multiple virtual environments.

A difference between these approaches is associated with their performances. As discussed in the previous paragraph, Docker relies on containerization, while Vagrant utilizes virtualization. In this latter approach, each virtual machine runs its own entire operating system inside a simulated hardware environment provided by special programs. Thus, each virtual machine needs a dedicated amount of static resources (CPU, RAM, storage), generating an overhead of such resources. Approaches based on containerization present a higher performance since containers simply use whatever resources it needs. This means, there is not overhead of resources. Based on this discussion, Docker is lighter than Vangrant. A deeper study in such approaches show that both have advantages and disadvantages, so that the final decision must be based on the particular features of each project.

There is another important difference between these approaches. Vagrant cannot create virtual machines or containers without virtualization platforms (Younge et al., 2011) such as VirtualBox, VMware or Docker. Differently, Docker can work without Vagrant. In order, the main advantage of vagrant is that it provides an easy mechanism to reproduce environments. These frameworks can also be used together with configuration management tools/languages to implement more powerful IaC environments. Some examples are given in the next section.

2.4 Tools in Practice

The previous section showed that there are several options regarding frameworks and configuration management tools to support the implementation of the infrastructure as code principles. However, the literature presents few contributions regarding their practical use and the focus of this literature is on the specification of extensions that could improve the limitations of current tools rather than descriptions of real case studies. The work of Hüttermann (2012), for example, integrates Vagrant and Puppet and uses them to create a topology for IaC consisting of Vagrant and Puppet artefacts that are continuously built and stored in a version control system. While Vagrant allows the building of lightweight and portable virtual environments, based on a simple textual description; Puppet uses a declarative syntax to describe the desired state of a target environment and allows this description to be executed to create that state on a target machine. Hummer *et al* (2013) propose and evaluate a model-based testing framework for IaC, where an abstracted system model is used to derive state transition graphs. The resulting graph is then used to derive test cases. Their prototype extends the Chef IaC tool. However the authors comment that their approach is general and could be applied to other tools, such as Puppet. The work of Artac *et al* (2017) discusses several technologies involved in supporting IaC. Its main focus is on the OASIS TOSCA, which is an industrial practice language for automated deployment of technology independent and multi-cloud compliant applications.

In order, the majority of examples regarding IaC are focused on Cloud environment and they are related to specific features of such domain. For example, Zhu *et al* (2014) report results from experiments on reliability issues of cloud infrastructure and trade-offs between using heavily-baked and lightly-baked images. Their experiments were based on Amazon Web Service (AWS) OpsWorks APIs (Application Programming Interfaces) and they also used the Chef configuration management tool. Several other works regarding IaC in the Cloud domain are discussed in the literature, such as in (Bruneo et al., 2014; Scheuner et al., 2014).

The work of Spinellis (2012) is another example of study that discusses popular tools in the DevOps domain, which include CFEngine, Puppet and Chef. This work stresses the main function of such tools, which is to automate a system's configuration so that users write rules expressing how an IT system is to be configured and the tool will set up the system accordingly. Wettinger *et al* (2014) also shows that the DevOps community focuses on providing pragmatic solutions for the automation of application deployment. Then, the communities affiliated with some of the DevOps tools, such as Chef or Puppet, to provide artefacts to build deployment plans for certain application tasks. Thus, these two previous works (Spinellis, 2012; Wettinger et al., 2014) confirm the trend to some specific tools (Chef and Puppet) and their relation to aspects of automation. Unfortunately, the scientific literature does not discuss the use and evaluation of such tools in a DevOps context, considering real development cases. This is the major contribution of our work, as detailed in the next sections.

3 DEVOPS IMPLEMENTATION: A CASE STUDY

This section is divided into four parts. We first describe the object of this case study, which is a real application that we call *Xsolution* (pseudo name due to commercial issues). Next we describe the original strategy to deploy this application and the metrics that characterize the problems of such strategy.

Then, we present the implementation of our infrastructure as code approach, which is based on the *PowerShell DCS*, and how this new strategy significantly improved our deployment process. Finally, we stress the advantages of this approach when it is compared to other ways to implement infrastructure as code solutions, such as Chef and Puppet.

3.1 The Object of Study

Xsolution is a client-server solution that requires the deployment of a server and mobile modules to execute. The abstract architecture of this application is illustrated in Figure 1.

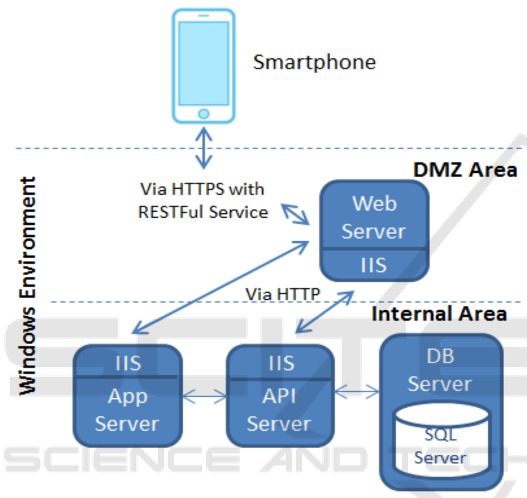


Figure 1: High level architecture of *Xsolution*.

Each of the components in this figure (Smartphone, Web server, Internet Information Service – IIS, App Server, API Server, DB Server and SQL Server) requires a specific configuration before the deployment of the application. This configuration used to be manually carried out by the IT team by means of an internal home-made deployment guide that describes all the process (step-by-step), as better detailed in Section 3.2.

In order, to prepare the required resources that will support Xsolution, or any other application with this architecture (Figure 1), the next actions must be carried out: (1) Installation of packages; (2) Database installation; (3) Installation of Web application requirements; (4) Installation of Web application; (5) Configuration of the Admin Web Applications; (6) Configuration of the log of errors; and (7) Mobile Web site configuration. Each of these actions has multiple steps and the traditional approach to carry out this process is to follow guides that describe these steps. This approach is described in the next section.

3.2 Manual Deployment Process

The manual deployment of Xsolution and other applications of our company, used to be manually carried out by a group of IT collaborators. In this strategy, each application had an associated deployment guide, which describes all the details to prepare the resources and environment to run this application. The internal deployment guide of *Xsolution*, for example, is a document with about 60 pages. It is important to understand how this manual process used to be carried out, so that we could have an idea about its complexity and the reasons it is a so time-consuming and error-prone activity.

The first step in this manual process is the installation of packages. Basically the idea is to create the directory structure, which will contain the admin front-end Web build files (related to user interface configuration), admin back-end Web build files, mobile android application, back-end mobile build files, database structure creation scripts, database initial seed script, and mobile user front-end Web build files. The second step is the database installation. *Xsolution*, for example, supports both Active Directory users (through Windows authentication) and SQL Server users (with custom login and password). The deployment team must also configure the IIS (Internet Information Service) to delegate the anonymous authentication configuration to *Xsolution*. However the main aim of this step is the creation of the database structure, which involves several details. For example, the structure must only be created in the first application deployment and the database scripts depend on the country where the application will be host. In fact there are a significant number of details that must be observed in this process. These details are descripted in the guide, such as:

“*if you update the database adding more values for some Enumeration, you must perform the Recycle of the Application Pools related to the App Server and API Server. This is necessary because the Enumerations present in this table are cached in memory when the application starts, rather than updated if changes were made in the database.*”

This type of conditional actions increases the complexity of the configuration and they are usually common source of errors since they are not part of the normal configuration flow. The use of further support tools, such as the SQL Server management studio to support the database backup procedures, is also described in the guide. In addition, there are also issues when databases are updated. For example:

"If you are upgrading the version of the database, you must sequentially run all scripts of the current version to the version you want. If just a script is ignored, the next scripts after that may not run correctly."

The third step is the installation of the Web application requirements. This step generally involves the installation of several third party resources, which act as the Front-end Admin Web, Back-end Admin Web and Mobile User Web. For example, the Xsolution requires the installation of the next components: (1) Windows Server 2008 R2 Service Pack 1 or Windows Server 2016; (2) NET Framework 4.5; (3) Internet Information Services 7.5 or 10; (4) ASP.NET; (5) Windows Management Framework 3.0; and (6) IIS URL Rewrite 2.0 module. Each of these components also has their own installation details, which must be observed by the deployment team. For example, the IIS module has its own manual (24 pages) with instructions about the reverse proxy configuration using an IIS server. One of the functions of the IIS is to capture the application log. This task is customized and also presents a set of configurations to properly work according to the features of each Web application.

The version of components is another point to observe. Xsolution, for example, allows the use of Windows Server 2008 R2 Service Pack 1 or Windows Server 2016. Depending on the choice, particular details must be observed along the configuration process. The configuration process also has influence of local laws. For example, due to the new national legislation for Internet (Law No. 12,965 - Internet Civil Landmark) (Tomasevicius Filho, 2016), information about the user access to the application needs to be stored for a period of six months. The information required is the IP, the user name, the date and time of login. Thus, the components must be configured to maintain such information.

The fourth step is the installation of the own Web application, which involves the creation of the application pool, the choice of Website locations and the assignment of each site to a specific application pool. In order, application pools are processing groups based on specific administrative preferences that isolate Website processes from other website processes on the server, offering strong performance and security benefits. Again, there are several details in this configuration. For example, the Admin Front-end Web and Admin Back-end Web applications could be in the same application pool, but it is strongly recommend that the Back-end Mobile application stay in a separate application pool. Thus, the configuration of two Web servers is required.

The fifth step is the configuration of the Admin Web application. There are several technical details in this step, which are related to authentication options, configuration of mobile responses and database access permissions. In fact, there are a significant number of parameters (about 50) that must be set and the deployment team must understand these parameters and know the best way to set them.

Finally, the sixth and seventh steps are respectively related to the configuration of the error log and mobile Website. Similarly to the other steps, the guide brings several details and customization options.

This description illustrates just part of the tasks and details regarding the manual deployment process. We can easily observe that this process is prone to errors, since it is long and has several details. Furthermore, it is hard to identify which configuration was not properly performed when an error occurs.

To demonstrate these problems and characterise this process in terms of software engineering metrics, we carried out a simple quantitative analysis of this process using *Xsolution* as our object of study. According to the schedule and documents from the *Xsolution* project, the deployment stage of each *Xsolution* release took about 16 hours in the best case. This means, when the process was performed without errors. Then, if we had 3 sprints per month, a collaborator should be allocated to this task over 6 days (8 hours/day) to each new version.

At each new sprint, all the guide items were executed, starting from the first step; while the own guide was also reviewed or updated along each sprint. This ensures a current and future process free of failures. If any error was identified, all the process was again started from the initial configuration. Thus, the final deployment could spend much more than 16 hours.

3.3 Infrastructure as Code Deployment

The infrastructure as code to support the deployment was implemented in our organization as a form to avoid the limitations of the previous manual approach (Section 3.2). Furthermore, this approach allows that solutions can be deployed in any environment without the expertise required by the manual approach.

Our strategy is based on the PowerShell DSC (Desired State Configuration), which is a script language that enables the definition of a set of deployment actions. Our experiments showed that several of the previous deployment actions could be automated with this language, such as: (1) install or remove server roles and features; (2) manage registry settings; (3) manage files and directories; (4) start, stop, and manage processes and services; (5) manage local groups and user accounts; (6) install and manage packages such as .msi and .exe; (7) manage environment variables; (8) fix a configuration that has drifted away from the desired state; and (8) discover the actual configuration state on a given node. Furthermore, DSC is a platform build into Windows, so that it is a natural choice to development projects in such platform.

The use of PowerShell DSC involved three phases in our experiments. In the first phase (authoring phase), the DSC configuration was created by means of the PowerShell Integrated Scripting Environment (ISE), which is an authoring tool for DSC configurations. These configurations are translated to one or more MOF files, which contain the necessary information for the configuration of the nodes. MOF (Managed Object Format) is a schema description language used for specifying the interface of managed resources, such as storage, networking, etc.). The MOF files are basically made up of a series of class and instance declarations, such as the next example (Figure 2):

```
#pragma namespace ("\\\\.\\Root\\Example1")
[ abstract,
   Description("This class defines common characteristics of Storage devices")
]
class MySchema_Storage
{
   [ read, key,
      Description("The device ID must be unique for each storage instance"):
         DisableOverride ToSubClass ]
   uint64 DeviceId;
   // Device Id is an unsigned integer 8 bytes
};
[ Description("The CD_Rom class describes characteristics of CDROM drives "):
   ToSubClass
]
class MySchema_CDROM:MySchema_Storage
{
   [write (true): ToSubClass]
   string ManufacturerName;
   [write (true): ToSubClass]
   string Model;
   [read(true), Description("This property contains the read speed of "
      "the CDROM. Example: 32"): ToSubClass ]
   uint16 ReadSpeed;
   [ write, Description("This property defines the BIOS revision of the CDROM."
      "Example: 7.13.1200"): ToSubClass ]
   string BIOSVersion;
};
```

Figure 2: MOF file example to a storage resource. Source: [http://www.informit.com/articles/article.aspx?p=30482&seqNum=10].

Next example (Figure 3) shows part of a MOF file used in the *Xsolution* deployment, which accounts for the configuration of roles and service roles during the installation of the Web application requirements (step 3 discussed in Section 3.2). A server role is a set of software programs that, when installed and properly configured, allows a computer to perform a specific function for multiple users or other computers within a network. Role services are software programs that provide the functionality of a role. In the manual way, the deployment team must access different configuration pages and check a set of options indicated by the manual. The next code automatically identifies the parameter to be configured and apply the indicated configuration.

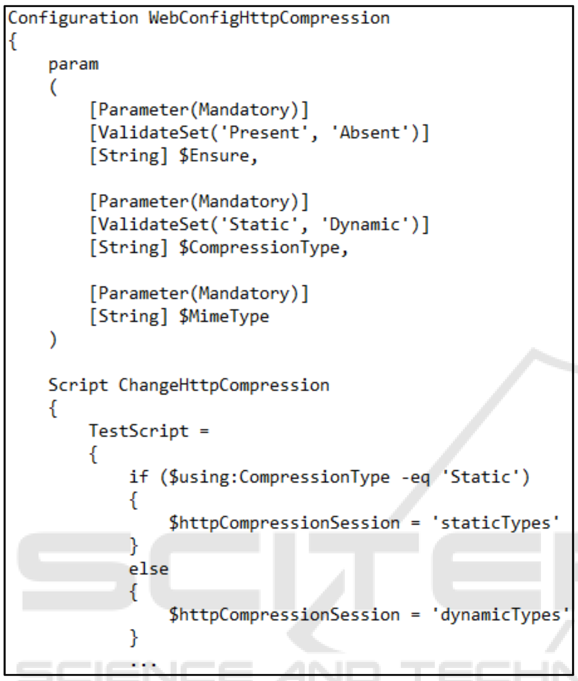
```
Configuration IisFeatures
{
    foreach ($feature in @('NET-Framework-45-Core', 'NET-
             Framework-45-ASPNET', 'Web-Net-Ext45', 'Web-Asp-
             Net45', 'Web-Default-Doc', 'Web-Static-Content',
             'Web-Http-Logging', 'Web-Request-Monitor', 'Web-Http-
             Tracing', 'Web-Filtering', 'Web-Stat-Compression',
             'Web-Dyn-Compression'))
    {
        WindowsFeature "IisFeature-$feature"
        {
            Ensure = 'Present'
            Name   = $feature
        }
    }
```

Figure 3: DSC script to configure the current state of a role and its service roles.

This script is simple and powerful at the same time since we do not need to indicate any path for the system variables. The own DSC framework already identifies such variables and set them. This process is completely transparent to human operators. However this configuration was simple because the DSC framework has the "WindowsFeature" as one of its 12 built-in configuration resources. In order, a DSC resource is a Windows PowerShell module, which contains both the schema (the definition of the configurable properties) and the implementation (the code that does the actual work specified by a configuration) for the resource. A DSC resource schema can be defined in a MOF file, and the implementation is performed by a script module. Other examples of built-in resources that were used in our study are:

- DSC File Resource: provides a mechanism to manage files and folders on the target node;
- DSC Package Resource: provides a mechanism to install or uninstall packages, such as Windows Installer and setup.exe packages, on a target node;
- DSC Service Resource: provides a mechanism to manage services on the target node.

Some of our required configurations were not provided as a built-in resource. However, the DSC framework supports the extension of such resources by means of classes, which defines a schema and its implementation. To evaluate this feature, we decided to implement a resource to configure the dynamic compression on mobile responses. Figure 4 shows the initial part of this implementation.

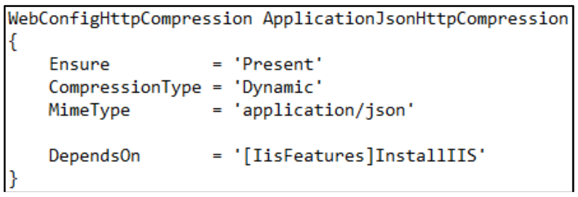


```
Configuration WebConfigHttpCompression
{
    param
    (
        [Parameter(Mandatory)]
        [ValidateSet('Present', 'Absent')]
        [String] $Ensure,

        [Parameter(Mandatory)]
        [ValidateSet('Static', 'Dynamic')]
        [String] $CompressionType,

        [Parameter(Mandatory)]
        [String] $MimeType
    )

    Script ChangeHttpCompression
    {
        TestScript =
        {
            if ($using:CompressionType -eq 'Static')
            {
                $httpCompressionSession = 'staticTypes'
            }
            else
            {
                $httpCompressionSession = 'dynamicTypes'
            }
            ...
```

Figure 4: Part of the implementation for a customized DSC resource to configure the http compression feature.

The first part of this resource (param) specifies the resource schema, which defines the parameters of the resource and its possible values. The second part defines the resource script, which implements the logic of the resource. After the resource creation, it was used in several parts of the configuration process by means of a simple resource call, such as in Figure 5. Again, the script avoids the search for the correct attributes in several configuration tabs and ensures that the desirable values are in fact set in the system.

The use of the infrastructure as code had a huge impact in our deployment efficiency. The deployment time for each release was decreased to 30 minutes. Thus, if we had 3 sprints per month, just 90 minutes will be spent in this process for each new version. Furthermore, all the process is automatic, so that it can be quickly executed from the beginning and the deployment team abandoned both the use of the guide (Section 3.2) and its update. Modifications are now carried out in the own scripts and maintained by version control programs.



```
WebConfigHttpCompression ApplicationJsonHttpCompression
{
    Ensure          = 'Present'
    CompressionType = 'Dynamic'
    MimeType        = 'application/json'

    DependsOn       = '[IisFeatures]InstallIIS'
}
```

Figure 5: Call of the WebConfig Http Compression resource.

3.4 Comparison to Other Approaches

The question that we intend to answer here is "why to use PowerShell DSC rather than other more popular approaches such as Chef and Puppet?". PowerShell DSC comes with the Windows OS by default, so that it is a good choice for managing Windows environments. While PowerShell DSC is able to directly access the Windows resources; Puppet and Chef requires an extra layer to access such resources. Chef, for example, started its support for Microsoft Windows from 2011 when it released the knife-windows plugin, which plays the role of this additional layer. Furthermore, Ruby must also be installed on Windows.

The use of Puppet is similar since it does not also have direct access to the Windows resources. Messages from Puppet users in specialised forums corroborate this affirmation. For example, "*We use Puppet in Windows. It works, but feels like a second-class platform*" (www.reddit.com). This means, the integration Puppet-Windows is not natural. Thus additional tools, such as *Chocolatey*, are available to facilitate this integration.

Even considering these tools, the deployment team commonly needs to implement additional recipes to improve this integration and access the Windows resources. Thus, the use of PowerShell DSC, considering the deployment to the Windows Platform, tends to be an easier and faster process.

There are some works that discuss the mutual use of Chef, or Puppet, and PowerShell DSC. The idea is to take advantage of the best features of each approach. The investigation of this hybrid strategy is one of the subjects for our future investigations.

4 LESSONS LEARNED

Some lessons were learned along our experience with DSC and some of them support previous finds from the literature.

DSC enables IT teams in deploying several time their configuration without risks of breaking the infrastructure. Thus, DSC in fact supports the DevOps principle of continuous deployment. We observed two important DSC features that optimize this process of continuous deployment:

- Only settings that do not match will be modified when the configuration is applied. The remainder configurations are skipped so that we obtain a faster deployment time;
- The definition of the configuration data and configuration logic are separated and well-defined. This strategy supports the reuse of configuration data for different resources, nodes and configurations.

A useful DSC strategy is to record errors and events in logs that can be viewed in the Event Viewer application. This function was important mainly at initial phases of the development, since the composition of configuration scripts was challenging for members of our team. Thus, the use of logs has facilitated the identification and solving of issues.

DSC provides a declarative syntax to express configurations for infrastructure and information systems. This DSC feature accounts for creating a transparent process, where the IT team do not necessarily have to know how DSC will provide a specific feature or software installation because the declarative syntax is similar to an INI type expression, specifying what should be present on the node, as discussed in (O'Connor, 2017).

DSC has two modes of operation: push and pull. The pull mode has its scalability as primary advantage and it seems to be the most used DSC mode. In fact, a single pull server can provide DSC configurations to many connected nodes with the additional benefit of specifying how often the LCM (Local Configuration Manager) on each node should check back with the pull server enforcing a configuration. However, as our task is focused on deployment, whose configuration is applied once for a long period, the push mode was chosen since we do not need periodic configuration checks.

Finally, we used the ability of DSC to create new resources to configurations that are not provided as a built-in resource. This process was straightforward and the resultant resources could be reused in several parts of the deployment script. Thus, this feature was very useful to a complete automation of our deployment process.

5 CONCLUSIONS

To the best of our knowledge, this work is the first to provide an initial analysis on the use and advantages of applying an infrastructure as code strategy to deployment, based on the PowerShell DSC. In fact, specialised forums and the software engineering community comment this lack. We could find comments such as "*I've never seen anyone with a robust production environment using DSC exclusively yet, however there are plenty of examples of Puppet/Chef environments*". Thus, this paper is a first contribution in this direction.

Our analysis was based on a case study, which used a real market application as object. The quantitative analysis of the efficiency of the approaches shows that the use of PowerShell DSC offers the appropriate resources to the automation of deployment process. However, our conclusions were based on solutions that run on the Windows platform. There are a few informal reports on the use of PowerShell DSC in Linux. However we cannot extend our conclusions to the Linux platform, since the particularities of this environment may bring a new set of challenges that must be analysed.

Our future researches intend to carry out a better quantitative analysis since the infrastructure as code is in fact being implemented in our organization. Thus, several quantitative and qualitative data is going to be generated regarding the real advantages of this deployment approach.

REFERENCES

Artac, M., Borovssak, T., Di Nitto, E., Guerriero, M., Tamburri, D., 2017. DevOps: Introducing Infrastructure-as-Code. *Proceedigns of the 39th IEEE International Conference on Software Engineering Companion*, pp. 497-498.

Bang, S. K., Chung, S., Choh, Y., Dupuis, M., 2013. A grounded theory analysis of modern Web applications: knowledge, skills, and abilities for DevOps. *Proceedings of the 2nd annual conference on Research in information technology* (RIIT '13). ACM, New York, NY, USA, pp. 61-62.

Bruneo, D., Fritz, T., Keidar-Barner, S., Leitner, P., Longo, F., Marquezan, C., Metzger, A., Pohl, K., Puliafito, A., Raz, D., Roth, A., Salant, E., Segall, I., Villari, M., Wolfsthal, Y., Woods, C., 2014. CloudWave: Where Adaptive Cloud Management Meets DevOps. *Proceedings of the IEEE Symposium on Computers and Communications*, pp.1–6. IEEE Press, New York.

Claps, G. G., Svensson, R. B., Aurum, A., 2015. On the Journey to Continuous Deployment: Technical and

Social Challenges Along the Way. *Information and Software Technology*, 57(1):21-31.

DeGrandis, D., 2011. Devops: So you say you want a revolution? *Cutter Business Technology Journal,* 24.8, pp. 34-39.

Duvall, M. P., 2011. *Continuous Delivery Patterns and AntiPatterns in the Software LifeCycle*.

Dyck, A., Penners, R., Lichter, H., 2015. Towards Definitions for Release Engineering and DevOps. *Proceedings of the IEEE/ACM 3rd International Workshop on Release Engineering*.

Ebert, C., Gallardo, G., Hernantes, J., Serrano, N., 2016. DevOps. *IEEE Software*, vol. 33, no. 3, pp. 94-100.

Erich, F., Amrit, C., Daneva, M., 2014. *Report: Devops literature review*. University of Twente, Tech. Rep.

Hosono, S., 2012. A DevOps framework to shorten delivery time for cloud applications. *International Journal of Computational Science and Engineering* 7(4): 329–344.

Humble, J,. Farley, D., 2010. *Continuous delivery: reliable software releases through build, test, and deployment automation*, Addison-Wesley Professional.

Hummer, W., Rosenberg, F., Oliveira, F., Eilam, T., 2013. Testing Idempotence for Infrastructure as Code. Eyers, D., Schwan, K (eds.) *Middleware*. pp. 368-388. LNCS, vol. 8275, Springer Berlin Heidelberg.

Hüttermann, M., 2012. *Infrastructure as Code. In: DevOps for Developers*. Apress, Berkeley, CA.

Jones, D., Siddaway, R., Hicks, J., 2013. *PowerShell in depth: An administrator's guide*. Shelter Island, NY: Manning Publications.

Liu, Y., Li, C., Liu, W., 2014. Integrated Solution for Timely Delivery of Customer Change Requests: A Case Study of Using DevOps Approach. *International Journal of U-& E-Service, Science & Technology*, 7, 41–50, 2014.

Loukides, M., 2012. *What is DevOps? Infrastructure as Code*, O´Reilly Media, 2012.

Lwakatare, L. E., Kuvaja, P., Oivo, M., 2015. Dimensions of DevOps. Lassenius C., Dingsøyr T., Paasivaara M. (eds) *Agile Processes in Software Engineering and Extreme Programming*. Lecture Notes in Business Information Processing, pp. 212-217.

Miell, I., Sayers, A. H., 2016. *Docker in Practice*. Manning Publications Co.

O'Connor, R., Elger, P., Clarke, P., 2017. Continuous software engineering – a microservices architecture perspective. *Journal of Software: Evolution and Process*, 29 (11).

Peacock, M., 2015. *Creating Development Environments with Vagrant*. Packt Publishing Ltd.

Punjabi, R., Bajaj, R., 2016. User stories to user reality: A DevOps approach for the cloud. *IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology* (RTEICT), Bangalore, pp. 658-662.

Scheuner, J., Leitner, P., Cito, J., Gall, H., 2014. Cloud Work Bench--Infrastructure-as-Code Based Cloud Benchmarking. *Proceedings of the IEEE 6th International Conference on Cloud Computing Technology and Science* (CloudCom), pp. 246-253.

Scheepers, M. J., 2014. Virtualization and containerization of application infrastructure: A comparison. *Proceedings of the 21st Twente Student Conference on IT*, pp. 1-7.

Shang, W., 2012. Bridging the Divide between Software Developers and Operators using Logs. *Proceedings of the 34th International Conference on Software Engineering*, pp. 1583–1586. IEEE Press, New York.

Spinellis, D., 2012. Don't install software by hand. *IEEE Software*, 29(4), 86-87.

Tessem B., Iden, J., 2008. Cooperation between developers and operations in software engineering projects. *Proceedings of the 2008 International Workshop on Cooperative and Human Aspects of Software Engineering*, pp. 105-108.

Tomasevicius Filho, E., 2016. Marco Civil da Internet: uma lei sem conteúdo normativo. *Estudos Avançados*, 30(86): 269-285. (*in Portuguese*)

Walls. M., 2013. Building a DevOps Culture. Sebastopol, CA: O'Reilly Media.

Wettinger, J., Andrikopoulos, V., Strauch, S., Leymann, F., 2014. Characterizing and Evaluating Different Deployment Approaches for Cloud Applications. *IEEE International Conference on Cloud Engineering*, pp. 205–214. IEEE Press, New York.

Wettinger, J., Breitenbücher, U., Leymann, F., 2014. DevOpSlang – Bridging the Gap between Development and Operations. Villari M., Zimmermann W., Lau KK. (eds) *Service-Oriented and Cloud Computing*. ESOCC 2014. Lecture Notes in Computer Science, vol 8745. Springer, Berlin, Heidelberg.

Younge, A. J., Henschel, R., Brown, J. T., Von Laszewski, G., Qiu J., Fox, G. C., 2011. Analysis of virtualization technologies for high performance computing environments. *IEEE International Conference on Cloud Computing*, pp. 9-16.

Zhu, L., Xu, D., Xu, X., Tran, A. B., Weber, I., Bass, L., 2014. Challenges in Practicing High Frequency Releases in Cloud Environments. Proceedings of the 2nd International Workshop on Release Engineering, pp. 21–24, Mountain View, USA.