A Computing Environment Configuration Management Pattern based on a Software Product Line Engineering Method

Alessandro Ferreira Leite, Safran
Diana Penciuc, IRT SystemX

This paper describes a pattern to configure computing environments based on a software product line engineering (SPLE) method. Configuring computing environment represents a challenging and time-consuming activity, even for skilled DevOps engineers. The challenges these users usually face include: (a) choosing a configuration management tool to write their configuration management scripts, (b) ensuring that their computing environments are correctly configured, (c) keeping configuration scripts' dependencies and relationships up-to-date, and (d) ensuring that their scripts are both reproducible and idempotent. Furthermore, configuration management tools offer different levels of abstraction to describe the tasks. Hence, they demand knowledge of various programming languages. Therefore, configuring a computing environment follows a pattern. The pattern is: (a) describe a target state for the computing environment, (b) identify the software packages and their required configuration files, (c) create the scripts with the commands to achieve the desired state, and then, (d) execute the scripts. Thus, a software product line (SPL) based strategy is ideal for this domain, as the products have common characteristics and variable parts. As a result, this approach demands much less time and effort than the traditional one.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Evaluation/ methodology*

General Terms: Design pattern, Computing environment configuration

Additional Key Words and Phrases: Infrastructure as Code, DevOps, Software Product Line

1. CONTEXT

Modern computing systems may comprise tens or even hundred of infrastructure elements. In this context, a recurring question is how to manage these elements efficiently. It means (a) to have a clear understanding of environment changes, which includes software packages and computing resources (e.g., virtual machines, storages, and networks); (b) to be aware of the changes that have been applied correctly in order to try to guarantee consistent system's state; and (c) to known how to rollback in case of failures [Woods 2016]. Indeed, this represents a challenging task, specially in the context of cloud computing environment, where it is easy to add and to remove computing resources [Buyya et al. 2009]. Consequently, the concept of infrastructure as code (IaC) has come out to help teams dealing with these issues [Hummer et al. 2013]. Infrastructure as code comprises a set of practices from software development to automate the provisioning, configuration, and management of computing environments. It aims to improve repeatability of computing environment states [Hüttermann 2012]. As a result, DevOps[1] teams rely on configuration management tools to both describe and provision the resources, as well as to change their states [Humble and Molesky 2011; Bass et al. 2015]. DevOps intend to reduce the separation between developers and operations, aiming to reduce the time between committing a change to a system in a version control system (VCS) and its place into a production environment based on an automated deployment process [Bass et al. 2015]. Provision means making computing resources ready to use. Computing resources include physical or virtual machines, storage, and network. Hence, the work depends on the type of resource.

Nowadays there exist various configuration management tools to aid on achieving this objective. Examples of tools include Ansible (*ansible.com*), Docker (*docker.com*), Chef (*chef.io*), Puppet (*puppet.com*), Vagrant (*va-*

[1] Short for development and operations

grantup.com), among others. Technically, these tools allow the teams to write the desired state for a computing through configuration management scripts [Hummer et al. 2013].

A script comprehends a sequence of actions to modify the states of computing resources. On the one hand, a **state** consists of (a) software packages that must be installed; (b) network configuration, including inbound and outbound traffic rules; (c) users' access permissions; and (d) services that must be running on a computing environment. On the other hand, **actions** comprise the commands to execute on the system to achieve the desired state. Generally, they include the name of the software packages to be installed via a package management system (e.g., apt, yum, and brew), as well as the corresponding command (e.g., install, update, remove). For example, a state description may include: *a virtual machine with **2 CPU cores** and* 2 GB ***of RAM memory** running **Ubuntu 14.04** and **WordPress** accessible through **HTTP** and **HTTPS** protocols provided by **nginx***. To achieve this objective, we can use *Vagrant* as the configuration management tools to spin up a virtual machine and to configure it executing the script illustrated in Figure 1. In this case, the actions are (i) load an operating system (e.g., Ubuntu 14.04); (ii) install a web server (i.e., nginx), (iii) install a database server (i.e., MySQL server) and a PHP engine required by WordPress; (iv) create a user and a database for WordPress on MySQL; (v) configure nginx to redirect PHP scripts' calls to the PHP engine; and (vi) start the applications on the machine.

2. PROBLEM

Configuration management tools target scripts execution, leaving for users the work of creating and combining various configuration management scripts. Moreover, they leave for users the work of maintaining scripts' dependencies and relationships up-to-date, which adds a new layer of complexity. Likewise, writing and executing configuration management scripts represents a time-consuming and error-prone activity even for experienced DevOps engineers.

Hence, some of the problems related to computing environment configurations are:

(a) **failure handling**: unless the configuration management tool supports *convergent deployment automation*, the engineers are responsible for manually restart the configuration process when a script fails. Convergent deployment means that if an error occurs during the execution of a script, and the expected system's state is not reached, the script is automatically re-executed until the system's state converge toward the desired one [Wettinger et al. 2014]. On the one hand, adopting a tool with this feature frequently leads to vendor lock-in. On the other hand, implementing convergent deployment is often time-consuming, and it increases the complexity of writing the configuration management scripts, as they must be composed of idempotent actions.

(b) **low-level of reuse**: configuration management tools have different domain-specific language (DSL). Even infrastructure and template definition tools such as CloudFormation (*aws.amazon.com/cloudformation*), Heat (*wiki.openstack.org/wiki/Heat*), Packer (*packer.io*), and Terraform (*terraform.io*) have their languages to describe high-level infrastructure resources and their relationships. As a result, combining scripts from different configuration management tools result in duplicate code. This occurs, since the configuration tools follow a procedural approach, which means that the engineers are responsible for defining the work to do and how it should be done. Furthermore, they have to know the state of the system before the execution of the scripts. For example, considering that in the scenario described in Section 1, the users decide to use (a) *Vagrant* to provision the virtual machine; (b) *Ansible* to install and configure *nginx*; and (c) Docker to run WordPress, MySQL, and PHP engine as Linux containers [Merkel 2014; Pahl and Lee 2015], as illustrated in Figure 2. These changes demand the engineers to write different configuration scripts (Figure 3) and to coordinate their executions. Likewise, the engineers are also responsible for guaranteeing that the scripts chaining leads to the required state.

(c) **guarantee consistent systems' state**: having the configuration scripts do not ensure that running them leads to a correct system state, as unexpected constraints might have been introduced by third dependencies, for instance. In other words, having the script does not guarantee reproducible environments states. A

concrete example includes Node.js that recently changed its package management address, as can be seen in [NodeJS 2016]. Thus, this change will soon break[2] the configuration scripts that use the previous address. Another example includes the changes in the general purpose Amazon Elastic Compute Cloud (EC2) instance types (*aws.amazon.com/ec2/instance-types*) that now require from the users to firstly configure a virtual private cloud (VPC) (*aws.amazon.com/vpc*) network to be able to create a virtual machine (VM). Hence, this has broken existing configuration management scripts.

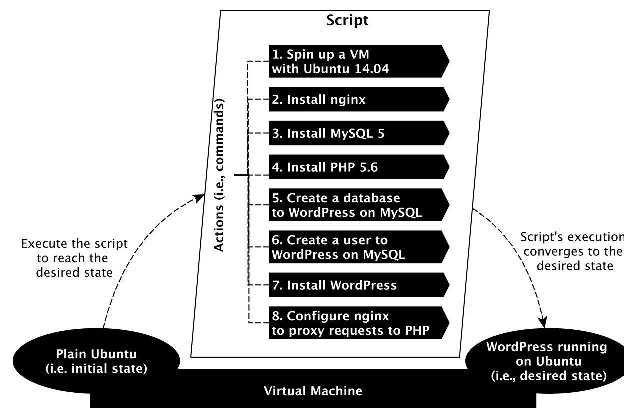

Fig. 1: Example of a script with the actions to install WordPress in a machine

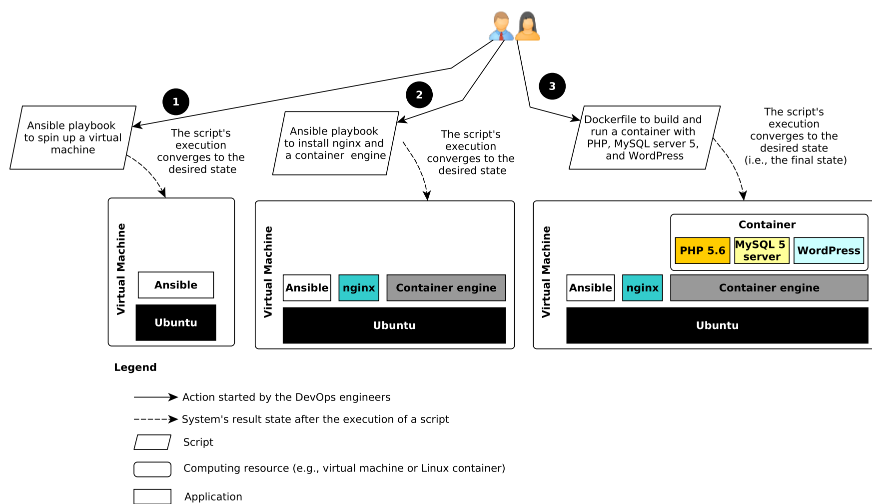

Fig. 2: Combining different configuration management tools to run WordPress in a machine

[2]Nowadays, Node.js shows the following message when referencing the old address. "*...you should migrate to a supported version of Node.js as soon as possible ...*" (*bit.ly/2clS8GH*). Thus, the users have to test their configuration scripts regularly to identify this kind of change.

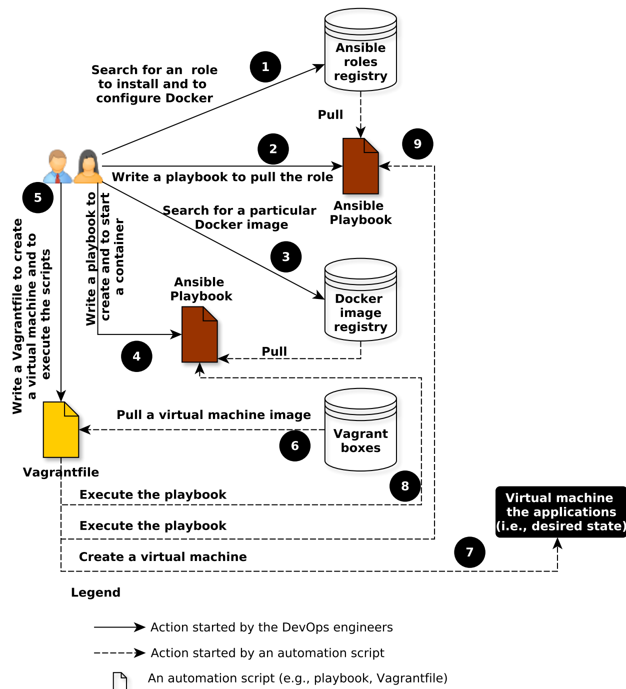

Fig. 3: Using Ansible, Docker, and Vagrant to provision a computing environment

3. SOLUTION

The solution comprises on embracing a feature-based strategy, where the focus shifts from writing and maintaining scripts to describe features. Thus, the configuration scripts and the computing environments become both products of a software product line.

A software product line (SPL) is a strategy to design a family of related products with variations in features, and with a common architecture [Clements and Northrop 2001; Pohl et al. 2005]. A feature means a user requirement or a visible system's function [Kang et al. 1990]. Hence, SPLE helps on developing a platform and to use mass customization to create a group of similar products that differ from each other in some specific characteristics. These characteristics are called *variation points* and their possible values are known as *variants* [Pohl et al. 2005]. This process can rely on *abstract* and *concrete* feature models and on a *configuration knowledge (CK)* [Czarnecki and Eisenecker 2000].

On the one hand, a *feature model (FM)* [Kang et al. 1990; Czarnecki and Eisenecker 2000] consists of a tree, where each node represents a feature of a solution. Relationships between a parent (or compound) feature and its child features (i.e., sub-features) are categorized as: *mandatory*, *optional*, *or* (at least one-child feature must be selected when its parent feature is), and *alternative* (exactly one-child feature must be selected) [Czarnecki and Eisenecker 2000]. Besides these relationships, constraints can also be specified using propositional logic to express dependencies among the features.

On the other hand, *configuration knowledge (CK)* defines how each feature is instantiated, its requirements and post-conditions. In practice, this means that the *configuration knowledge (CK)* makes a mapping between features and the artifacts that implement them. Figure 4 illustrates the use of a *feature model* to describe the functional properties of a *virtual machine*. In this example, *Processor* is an *abstract feature*, while *Ivy Bridge* is a *concrete feature*. As can be seen, a valid configuration description includes *{Debian, Ivy Bridge, Dedicated, Ten GB, One hundred GB, Provisioned, and Server}*, whereas an invalid one comprehends *{Ivy Bridge, Dedicated, Ten GB, One*

hundred GB, Provisioned, and Server} as it does not specify an *Operating System*. Moreover, the constraints c_1 and c_2 indicate that selecting the feature *Bootstrap* reduces the configuration space to only the *Operating System*.

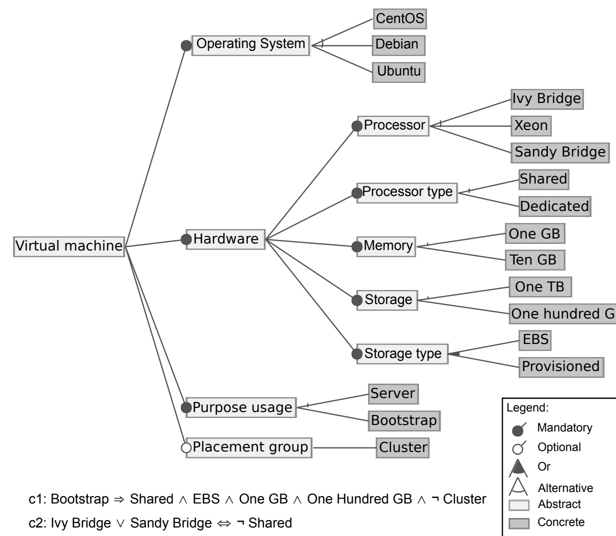

Fig. 4: Example of a feature model with *Optional*, *Mandatory*, *Or*, *Alternative* features, and two constraints

Thus, a SPLE helps on moving from a procedural strategy to a reactive one. In this context, Figure 5 presents a pattern that enables us to configure computing infrastructure following a product line approach. In this case, the activities of the pattern are:

(a) **create a repository of features**: in this activity, the DevOps engineers use feature model to describe the features, including their relationships and constraints. Feature model has the advantage of being technology independent, and it is normally understood by ordinary users. Figure 7 shows a feature model with the features to set up WordPress. Thus, selecting the features *Ubuntu* and *WordPress*, leaves to users the option to select a database, web server, and a PHP version. In this stage, it is also defined the product line. In other words, the assets and the outputs of the software product line, as depicted in Figure 6. In this context, the assets represent the configuration management tools, whereas the outputs comprise the artifact demanded by each tool.

(b) **define the configuration knowledge (CK)**: this step comprises the working of describing how each feature is instantiated, as well as in defining their binding points. Binding points are runtime informations that must be set by the system. This enables artifacts reuse and allow the engineers to delegate dependencies to be resolved at runtime.

(c) **instantiate the products**: this activity comprises the work of instantiating the products based on the users requirements. The products include the configuration management scripts and the computing environments. Thus, the engineers do not need to write scripts, but to generate them.

4. ACKNOWLEDGE

We would like to thank our shepherd, Fernando Lyardet, for his insightful comments that significantly improved this paper.

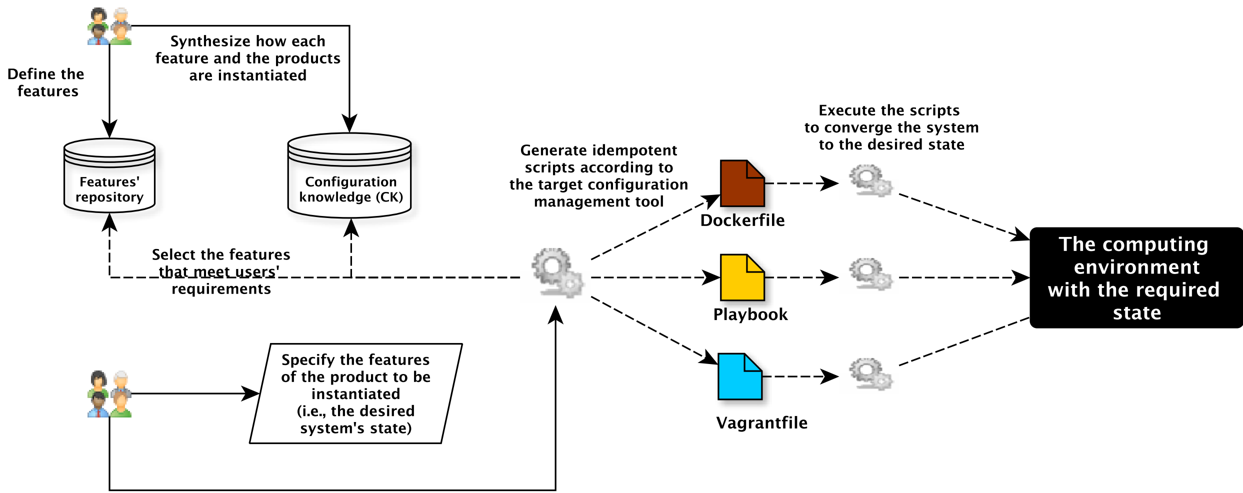

Fig. 5: A feature-based pattern to provision computing infrastructure

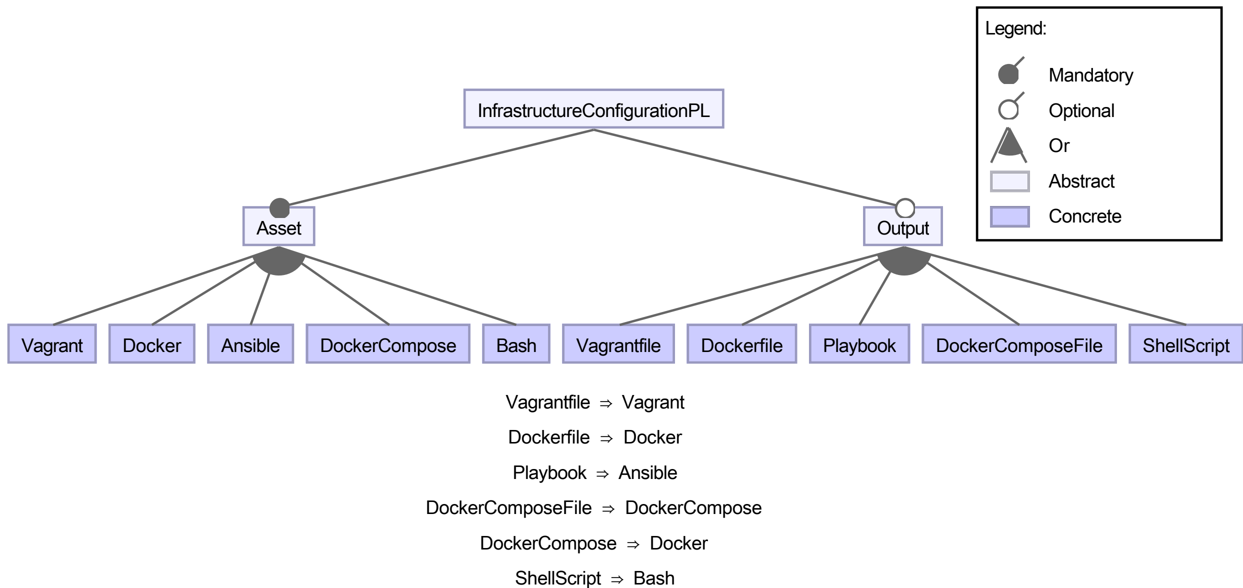

Fig. 6: Feature model representing the infrastructure configuration product line

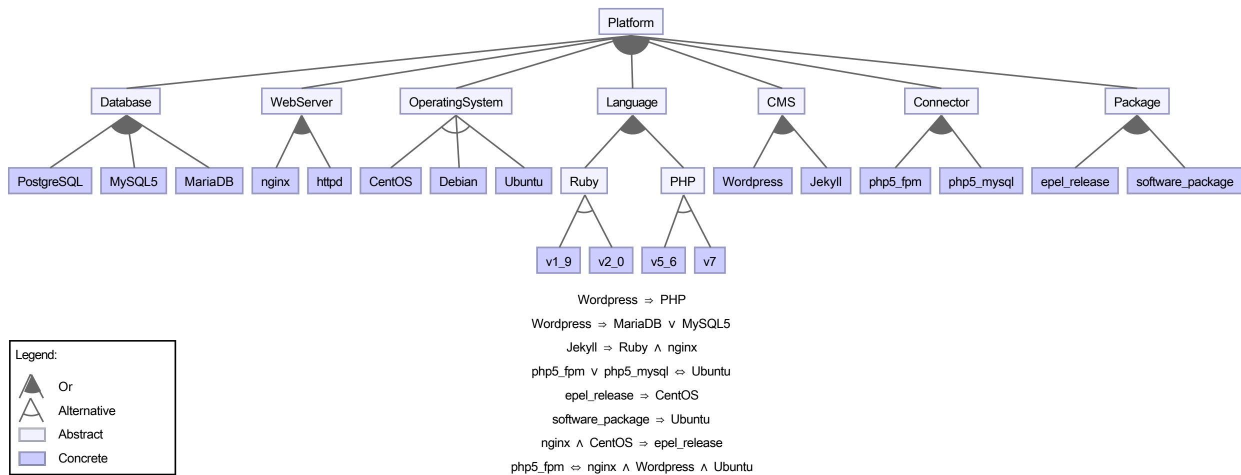

Fig. 7: Using feature model to describe the requirements and configuration options to set up a platform for a CMS

REFERENCES

BASS, L., WEBER, I., AND ZHU, L. 2015. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional.

BUYYA, R., YEO, C. S., VENUGOPAL, S., BROBERG, J., AND BRANDIC, I. 2009. Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems 25*, 599–616.

CLEMENTS, P. AND NORTHROP, L. 2001. *Software Product Lines: Practices and Patterns*. Addison-Wesley.

CZARNECKI, K. AND EISENECKER, U. W. 2000. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley.

HUMBLE, J. AND MOLESKY, J. 2011. Why enterprises must adopt devops to enable continuous delivery. *Cutter IT Journal 24,* 8, 6.

HUMMER, W., ROSENBERG, F., OLIVEIRA, F., AND EILAM, T. 2013. Testing idempotence for infrastructure as code. In *14th International Middleware Conference*. Springer Berlin Heidelberg, 368–388.

HÜTTERMANN, M. 2012. Infrastructure as code. In *DevOps for Developers*. Springer, 135–156.

KANG, K. C., COHEN, S. G., HESS, J. A., NOVAK, W. E., AND PETERSON, A. S. 1990. Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute. November. Last accessed in July 2016.

MERKEL, D. 2014. Docker: Lightweight linux containers for consistent development and deployment. *Linux Journal 2014,* 239.

NODEJS. 2016. Installing Node.js via package manager. Last accessed in September 2016.

PAHL, C. AND LEE, B. 2015. Containers and clusters for edge cloud architectures–a technology review. In *3rd International Conference on Future Internet of Things and Cloud*. 379–386.

POHL, K., BÖCKLE, G., AND LINDEN, F. J. V. D. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York.

WETTINGER, J., BREITENBÜCHER, U., AND LEYMANN, F. 2014. Compensation-based vs. convergent deployment automation for services operated in the cloud. In *12th International Conference on Service-Oriented Computing*. Springer-Verlag, 336–350.

WOODS, E. 2016. Operational: The forgotten architectural view. *IEEE Software 33,* 3, 20–23.

SugarLoafPLoP'16, November 16-18, Buenos Aires, Argentina. Copyright 2016 is held by the author(s). ACM 978-1-4503-0107-7