Cluster Comput
DOI 10.1007/s10586-017-0897-1

Dohko: an autonomic system for provision, configuration, and management of inter-cloud environments based on a software product line engineering method

Alessandro Ferreira Leite[1] · **Vander Alves**[1] · **Genaína Nunes Rodrigues**[1] · **Claude Tadonki**[2] · **Christine Eisenbeis**[3,4] · **Alba Cristina Magalhaes Alves de Melo**[1]

Received: 1 March 2017 / Accepted: 1 May 2017

Abstract Configuring and executing applications across multiple clouds is a challenging task due to the various terminologies used by the cloud providers. Therefore, we advocate the use of autonomic systems to do this work automatically. Thus, in this paper, we propose and evaluate *Dohko*, an autonomic and goal-oriented system for inter-cloud environments. *Dohko* implements self-configuration, self-healing, and context-awareness properties. Likewise, it relies on a hierarchical P2P overlay (a) to manage the virtual machines running on the clouds and (b) to deal with inter-cloud communication. Furthermore, it depends on a software product line engineering method to enable applications' deployment and reconfiguration, without requiring pre-configured virtual machine images. Experimental results show that *Dohko* can free the users from the duty of executing non-native cloud application on single and over many clouds. In particular, it tackles the lack of middleware prototypes that can support different scenarios when using simultaneous services from multiple clouds.

Keywords Autonomic system · Inter-cloud · Software product line engineering · Feature modeling

✉ Alessandro Ferreira Leite
alessandro.leite@acm.org

[1] Department of Computer Science, University of Brasilia, C.P. 4466, Brasilia 70910-900, Brazil

[2] MINES ParisTech / CRI, 35, rue Saint Honoré, 77305 Fontainebleau, France

[3] Inria / Université Paris-Sud 11, bâtiment 650, 91405 Orsay Cedex, France

[4] PCRI, bâtiment 650, 91405 Orsay Cedex, France

1 Introduction

Even though most of the cloud providers claim an infinite pool of resources, in practice even the biggest cloud provider may have scalability issues due to the increasing demand for computational resources by the users [72]. As a result, the clouds normally limit the number of resources that can be acquired in a period of time. Besides that, clouds may experience outage problems owing to regional problems such as network partition or technical problems like software bugs. Thus, relying on a single cloud represents a risk for the users, as well as it might result in significant performance degradation [10].

In this context, it is interesting to consider an *inter-cloud* environment. In an *inter-cloud*, a service or application runs on resources distributed across multiple clouds [25], and these clouds are unaware of each other [22]. Consequently, the users are responsible for aggregating the resources distributed across different clouds to execute their services or applications. Figure 1 illustrates an inter-cloud scenario, where two services run on resources distributed across three clouds.

In this case, the users have to: (a) identify the clouds and the virtual machine (VM) types that fit the applications needs; (b) select the virtual machine image (VMI) to launch the VMs on each cloud; and (c) deploy the applications considering functional and temporal dependencies, as well as clouds' constraints

Thus, deploying and executing applications on an inter-cloud scenario demand both cloud and system administration skills. Likewise, this work represents an error-prone and difficult task. The difficulties are mostly associated to the kind of applications considered by these clouds. While the clouds focus on web applications, the users' applications are many times batch-oriented, performing parameter sweep opera-

Published online: 23 May 2017

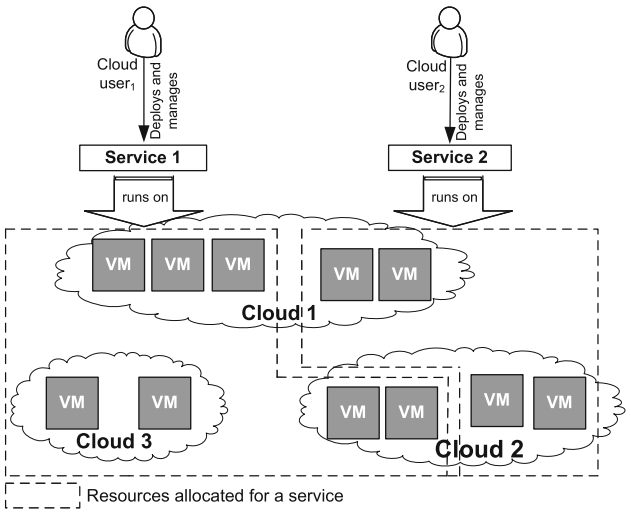


Fig. 1 Example of an inter-cloud scenario, where two different services run on resources distributed across three different clouds

tions. Furthermore, users applications may require specific configurations and some of them may take days or even weeks to complete their executions.

While the former issue can be dealt with pre-configured virtual machine image (VMI), the latter requires monitoring and fault-tolerance strategies in order to reduce the chances of losing the work just before it completes.

Even though virtual machine image can handle functional properties like minimum disk size, specific operating system, and particular software packages, it leads to a high number of configuration options. Hence, when the number of VMIs increases, the users face the issue of selecting one that meets their needs. Additionally, the usage of VMI usually results in vendor lock-in, and leaves to users the work of selecting the resources to deploy the images and to orchestrate them accordingly; i.e., the work of selecting and instantiating the virtual machine images in each cloud. Thus, this augments: (a) the deployment time, (b) the monetary cost, and (c) the maintenance efforts. In addition, image migration across multiple clouds has a high financial cost due to network traffic. Furthermore, in case of cloud's failure, it may be difficult for users to re-create the failed environment in another cloud, since the image will be inaccessible.

Recently, configuration management tools have evolved to enable a declarative specification of the desired state of computing environments. However, they still require both computer programming and system administration skills from the users. Likewise, they do not deal with automatic resource selection in a single or in an inter-cloud scenario. Examples of configuration management tools include Chef (*chef.io*), Ansible (*ansible.com*), Juju (*juju.ubuntu.com*), Puppet (*puppet.com*), and Vagrant (*vagrantup.com*).

Monitoring is usually realized by employing heartbeats [27]. Heartbeats are presence notification messages sent by each virtual machine to a monitor [13]. In this case, when a heartbeat message fails to reach the monitor within a given threshold, the VM is considered unavailable and a process to recovery it or to create a new one starts. Therefore, the users have to define and implement monitoring policies to identify virtual machines failures on each cloud and to restart them.

Clearly, an inter-cloud scenario highly benefits from automatic resource discovery mechanisms, automatic matching process, and homogeneous management interface. In practice, this means automated coordination of multiple cloud configurations. Indeed, automatic inter-cloud configuration adds a new dimension that currently lacks a successful technology/product. The difficulties are mostly due to clouds' heterogeneity and the lack of management tools to deal with inter-clouds.

In other words, inter-cloud environments require autonomic computing systems that can manage and adjust themselves according to users' goals [29]. Autonomic inter-cloud systems can free the users to concentrate on what they want to accomplish with the clouds rather than spending their time overseeing cloud's environments to get them there [29].

Therefore, in this paper, we propose and evaluate an autonomic and goal-oriented system for inter-cloud environments. Our system, namely Dohko, relies on a software product line engineering (SPLE) [12,52] method to implement (a) automatic resource discovery and selection; and (b) the autonomic properties [29]: self-configuration, self-healing, and context-awareness. Thus, following a declarative strategy, in *Dohko*, the users specify their applications and requirements, and it automatically: (a) selects the resources that meet the users' needs; (b) configures the whole computing environments on the clouds; (c) handles resource failures; and (d) executes the applications on the clouds. Additionally, our feature based method enables *Dohko* to work with the configuration management tools, including Ansible, Docker, and Vagrant. In this case, *Dohko* generates the deployment scripts and use the corresponding tool to execute them on the computing environment.

We executed a genomics application to compare up to 24 biological sequences with the UniProtKB/Swiss-Prot database in two different cloud providers—Amazon EC2 and Google Compute Engine (GCE)—, considering single and inter-cloud scenarios. Experimental results show that our system could transparently manage various clouds and configure the computing environments, requiring minimal human intervention. Moreover, by employing a hierarchical peer-to-peer (P2P) overlay to organize the nodes, our system could also handle failures and organize the nodes in a way that reduced inter-cloud communication. In particular, *Dohko* tackles the lack of middleware prototypes that can support various scenarios when executing services across multiple infrastructure-as-a-service (IaaS) clouds. Moreover,

it met some functional requirements identified for inter-cloud unaware systems [50] such as: (a) it can aggregate services from different clouds; (b) it provides a homogeneous interface to access the services from the clouds; (c) it implements automatic resource selection on the clouds; (d) it can deploy its components on multiple clouds; (e) it provides automatic procedures for deployments; (f) it utilizes an overlay network to connect and to organize the resources distributed across the clouds; and (g) it does not impose constraint for the selected clouds.

The reminder of this paper is organized as follows. Sect. 2 describes the challenges faced when working with multiple clouds. Next, Sect. 3 overviews key software product line engineering (SPLE) concepts used in our method. Then, Sect. 4 presents our SPLE method to enable resource provision and configuration in inter-cloud scenarios; i.e., to implement the self-configuration property. Section 5 presents the architecture of our system and its main components, followed by a description of its autonomic properties. Then, experimental results are discussed in Sect. 6. Next, a comparative view of some important features of cloud systems is discussed in Sect. 7. Finally, Sect. 8 concludes this papers and presents envisioned future works.

2 Challenges of inter-cloud environments

Over the years, the rapid development of cloud computing has pushed to a large market of cloud services, exposed through heterogeneous and proprietary application programming interfaces (APIs). This has resulted in distinct service descriptions, data representation, and message level naming conflicts, making service interoperability and portability a complex task [45,50].

In this context, some challenges related to the use of multiple clouds are:

(a) ***Resource discovery***: since each cloud provider employs different languages and formats to describe its services and resources, automatic resource discovery becomes a challenge. For example, Amazon EC2 uses Elastic Computing Unit (ECU) as a metric to express CPU capacity of a virtual machine, while GCE uses Google Compute Engine Units (GCEUS). In practice, these metrics cannot be compared or even converted from one to another. Although there exist some efforts to create standards for resource description [17,42,47,62], these standards are not yet supported by most of the cloud providers. Hence, to instantiate a computing environment correctly in a cloud, the users have to read extensive documentation. Consequently, considerable time and efforts are demanded from the users, even for experienced technical users.

(b) ***Resource selection***: appropriate resource selection requires detailed data. However, the lack of detailed data about cloud services performance criteria forces the users to select resources based on their experiences and on few data provided by the clouds. This occurs because the clouds employ high-level terms to describe the performance of their resources such as low, moderate, and high, limiting a decision based on imprecise resources' descriptions. Additionally, some descriptions demand specialized skills from the users. For example, Amazon EC2 offers instance types designed to deliver 10 Gbps of network throughput. However, this throughput is disabled by default and to enable it the users must: (a) install and activate a network driver in the instance's operating system; (b) stop the VM and enable a feature called *enhanced networking* (*aws.amazon.com/ec2/details*). In addition, the users must know that this feature is only allowed on *hardware-assisted virtualization* [1] and to enable it, they have to call a method of the EC2's API. As a result, the users must know a lot of detailed technical information about the instance types and the virtual machine images. Nevertheless, ordinary users lack such knowledge, which often leads to invalid configuration requests, increasing the configuration time and in some cases, the financial cost. Furthermore, clouds' guarantees are based on resource availability without performance commitment [32,49,66]. Typically, the providers expect to users benchmark their application(s) on the available resources to realize which one meets both performance and cost requirements. However, benchmarking demands a good knowledge about both applications' behaviors and resources' architecture. Furthermore, the providers have distinct cost models and pricing policies. For instance, some clouds do not charge data transfer between resources in the same region while, in others, this is only true in special cases. Clearly, as the number of cloud providers increases, the users have to deal with various clouds' and resources' prerequisites.

(c) ***Automatic configuration***: the typical option to deploy computing environments on cloud infrastructures relies on pre-configured virtual machine images. In this case, an image is configured and transfered to each cloud. Normally, image configuration is realized manually, following ad-hoc deployment scripts, or using a configuration management tool. In the manual approach, the users are in charge of executing all the commands to achieve a desired state. Clearly, this approach is error-prone, time-consuming, and it is commonly difficult to check if it achieved the desired state. Instead, in the deployment script strategy, the users write a set of imperative scripts, allowing it to be executed several times on different systems. Nonetheless, scripting still has most

of the drawbacks of the manual approach, as it may be executed in environments that do not meet the preconditions, leading to unpredictable outcomes. Recently, configuration management tools have evolved, allowing a declarative specification of the desired state. Examples of these tools are Chef (*chef.io*), Ansible (*ansible.com*), and Juju (*juju.ubuntu.com*). Nevertheless, these tools require both computer programming and system administration skills from the users, and they do not deal with automatic resource selection in single or inter-cloud scenarios.

3 Software product line engineering

Software product line engineering (SPLE) is a strategy to design a *family of related products* with variations in features, and with a common architecture [12,52]. A feature means a user requirement or a visible system's function [34]. 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* [52]. An SPLE process most often relies on *feature model* to define valid combinations of its assets [14].

A *feature model (FM)* [14,34] consists of a tree and constraints. In the tree, 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) [14]. Besides these relationships, constraints can also be specified using propositional logic to express dependencies among the features.

A feature model can be instantiated by selecting its features according to the aforementioned relationships and constraints in the model, leading to a *configuration*. In practice, since different groups and different people make configuration choices, this can be accomplished in stages [16]. In this work, we refer to the original feature model as an *abstract feature model* and to a partially instantiated one as a *concrete feature model*.

Feature models may also have attributes representing non-functional properties such as cost. Furthermore, feature models may use *cardinality* to express the relationships between the features. These relationships are classified as: *feature cardinality* and *group cardinality* [6]. *Feature cardinality* determines the number of instances of a feature that can be part of a product, and it is denoted as an interval $[n..m]$, where n and m represent respectively the lower and the upper bound. *Group cardinality*, on the other hand, limits the number of sub-features that can be included in a product when its parent feature is selected. One important characteristic of *feature models* is that they are commonly understood by non-technical users, since they refer to domain concepts [14,34].

4 Software product line engineering method

This section describes our method to tackle the three challenges presented in Sect. 2. It comprises a process, which is described in Sect. 4.1, an abstract model (Sect. 4.2), concrete feature models (Sect. 4.3), and a configuration knowledge (CK) (Sect. 4.4).

4.1 Process

Our process targets infrastructure-as-a-service (IaaS) inter-clouds, and it relies on: *abstract* and *concrete* feature models and on a *configuration knowledge (CK)* [14]. The *abstract feature model* provides a homogeneous and provider-independent resources' description, thus, enabling inter-cloud resource discovery (challenge (a)). The *concrete feature models*, on the other hand, are partial instances [15] of the *abstract model* representing clouds' configuration options. In this case, they enable resource selection at each cloud, which tackles the challenge (b). The configuration knowledge [14] comprises the data specifying how the products are instantiated (challenge (c)). *In our process, a product is a computing environment running on cloud infrastructures that meets users' and cloud configurations' constraints, including software package dependencies*.

Our process employs an off-the-shelf constraint satisfaction problem (CSP) solver to create a computing environment using resources distributed across various clouds. Thus, based on the *abstract feature model*, the solver selects: (i) the concrete models (i.e., the clouds) matching an environment description; (ii) the resources (e.g., VMs) at each model to instantiate, taking into account their dependencies; and (iii) the assets on the configuration knowledge.

Figure 2 shows our SPLE process. It has two phases named *domain engineering* and *product engineering*. The former comprises the activities that are performed at design time to develop reusable artifacts. In other words, the *domain engineering* phase describes the commonality and variability in the cloud domain, whereas the *product engineering* phase comprises the activities that bind the variabilities according to the users goals. The overall process fulfills with the domain and application engineering refinement proposed by [12] and [52].

Additionally, the process considers three distinct roles: *domain engineers*, *system engineers*, and *cloud users*. *Domain engineers* are cloud experts responsible for creating

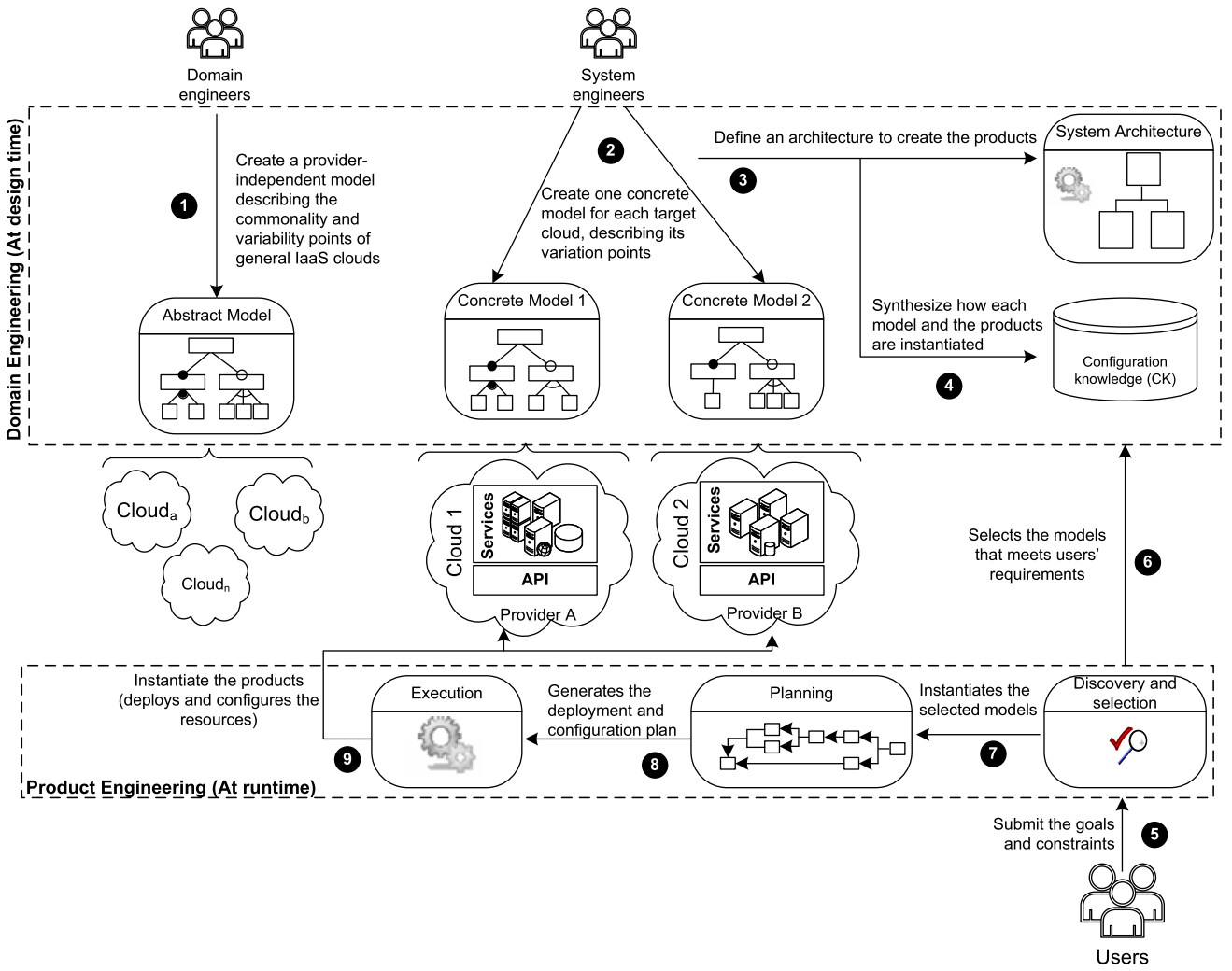


Fig. 2 The proposed SPLE process to enable automatic inter-cloud configuration

the *abstract model*. *System engineers* are *domain engineers* with *system administration* and *software architecture* skills. They are responsible for creating the *concrete models* and for defining the *configuration knowledge*. In other words, *system engineers* use the *abstract feature model* to bind the variation points of the clouds. Finally, *cloud users* are all people interested in using clouds' infrastructures.

In this context, when the users need to use the clouds, they submit their requirements, employing the terms defined in the *abstract model*. Then, our system looks for the *concrete models* that meet the users needs. Next, having the *concrete models*, the system creates the products, executing the actions described in the *configuration knowledge*. In this case, a computing environment is always a product of a model.

4.2 Abstract feature model

In general, the resources offered by IaaS clouds comprise: virtual machines, storage, networking, and virtual machine image (VMI). In practice, IaaS clouds provide the access to virtual machines, which demands the usage of other resources. Therefore, virtual machines are the principal service offered by IaaS clouds.

In our *abstract feature model*, an IaaS cloud is modeled as depicted in Fig. 3. It is important to highlight that to derive this model, we also considered the taxonomies available in the literature [28,53,63,77]. As can be seen, a *virtual machine* has an *instance type*, a *pricing model*, a *VMI*, and at least one *disk*. Additionally, a VM runs on an availability *zone* (i.e., data center), and it may belong to a *group*.

Instance types determine the *hardware configuration* of the VMs, and they are divided into different *family types*, which dictates the characteristics and the recommended usage for an *instance type*. This usage is based on the amount of resources available for the instances such as computing capacity (i.e., CPU), memory, and accelerators. Moreover, the members of a *family type* normally have the same hardware configuration, i.e., they are usually homogeneous with

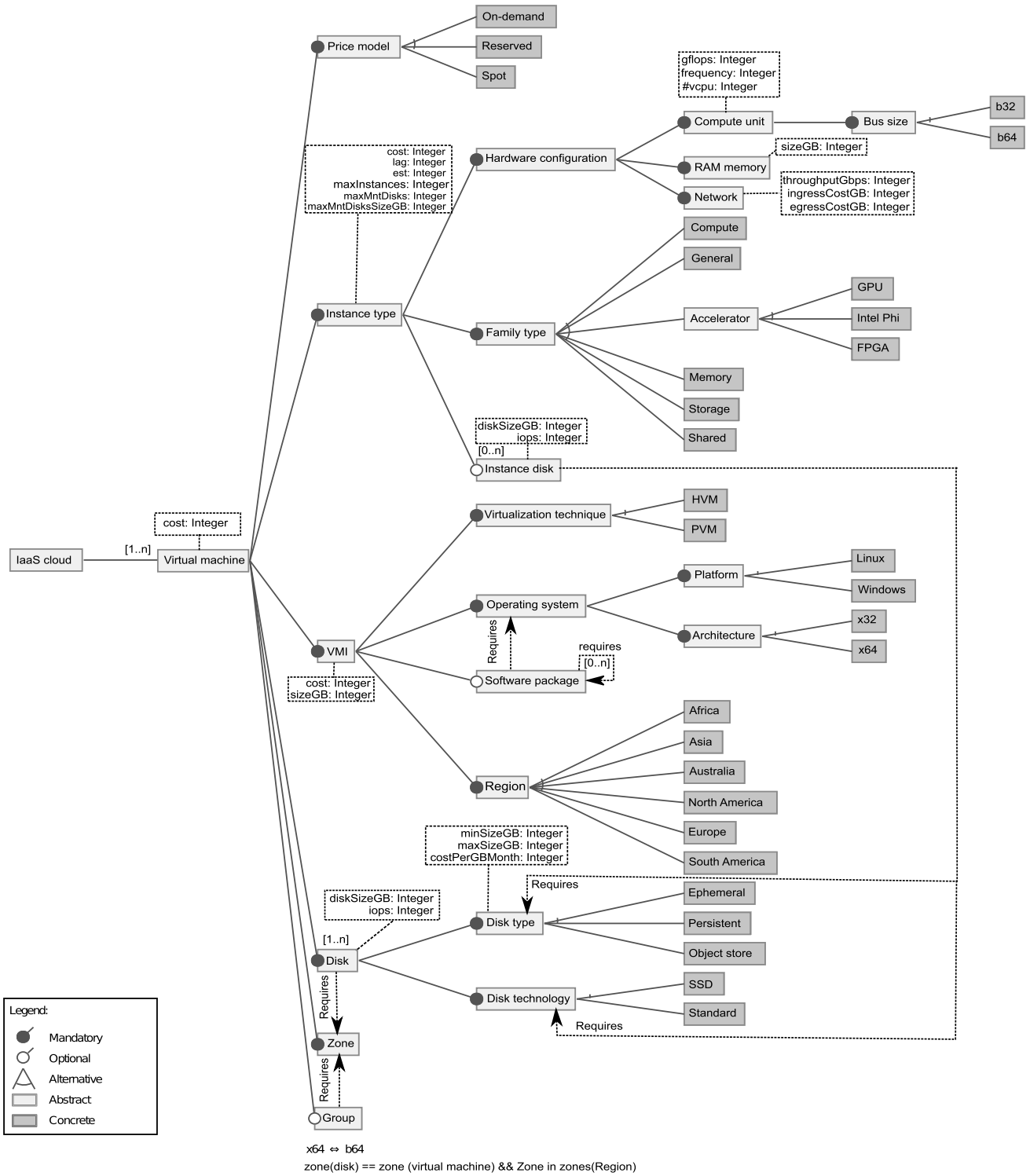


Fig. 3 Our abstract feature model

regard to the underlying infrastructure. With the exception of the *family type shared*, all the other *instance types* offer a fixed amount of resources. Furthermore, *instance types* that do not have a defined purpose usage are classified as *general*. For example, currently, Amazon EC2 offers seven *family types* (*aws.amazon.com/ec2/instance-types*) and Google Compute Engine offers four (*cloud.google.com/compute/docs/machine-types*).

Some providers and models [23,28,63] add a new level to classify the size of an *instance type*. In this case, they use

the terms *medium*, *large*, *xlarge*, among others to identity an *instance type*. However, these classifications differ only in the amount of resources provided by each *instance type*, which can be handled by using attributes (e.g., # of vcpu, memory size, among others). Furthermore, these classifications require additional data to describe an *instance type*, since VMs belonging to different *family types* may receive the same classification.

Instance types may also have *physical disks* attached to the host computer of a virtual machine, i.e., *instance disks*. These disks are normally *ephemeral* and they cannot be used as the primary *disk* of a virtual machine.

Every VM is instantiated from a virtual machine image, which maps the root file system, determines its minimal disk size, and the type of its hypervisor.

VMs can be placed in a *group* to decrease network latency and/or to increase network throughput, for instance. In this case, all VMs of a group run in the same *zone*. This feature enables us to reference to a *group* instead of for each virtual machine individually. Furthermore, it can also be used to model a physical cluster.

With our abstract model, we can see the kind of resources that are available in IaaS clouds. However, we cannot yet decide which cloud to use, as it does not specify what are the configuration options of each cloud, neither how to instantiate their features.

4.3 Concrete feature model

Figure 4 illustrates two *concrete feature models*, describing the configuration options of two distinct cloud providers—Amazon Elastic Compute Cloud (EC2) and Google Compute Engine (GCE). These models result from binding variability in the *abstract model* for each cloud, except the cardinality variability at the root. Moreover, they rely on well-known benchmarks, to report resources' performance. These benchmarks helped us to avoid vague or provider's specific terms. For instance, instead of saying that an *instance type* has a CPU performance of 88 ECUs, it is described as having 32 GFlops of sustainable performance. Hence, the benchmarks help on uniformizing performance descriptions, without forcing the providers to commit to a standard description.

In our proposal, CPU performance is obtained through the execution of LINPACK (*netlib.org/linpack*), which computes the sustainable performance metric. The UnixBench[1] (*github.com/kdlucas/byte-unixbench*) is employed to provide another way to compare the performance variations within the same cloud platform. In other words, the UnixBench benchmark enables us to compare the CPU performance of the instances that belong to the same *family type*, but with a variable number of virtual cores. For network performance, the iperf (http://software.es.net/iperf) application is used to measure TCP and UDP throughput.

Thus, through the concrete models (Fig. 4), we can see that both clouds offer resources in different regions—Europe and North America—, and that EC2's instance types have ephemeral SSD disks at zero cost, for example. Moreover, using these models, we can simulate computing environment migration between the clouds. In this case, we use the *abstract model* to describe the environment and the *concrete one* to select the configurations. For example, a description like: {*b64, #vcpu = 2, memorySizeGB = 15, General, HVM, Linux, x64, North America, Persistent, SSD, diskSizeGB = 30*} returns *VM1* of GCE (Fig. 4b).

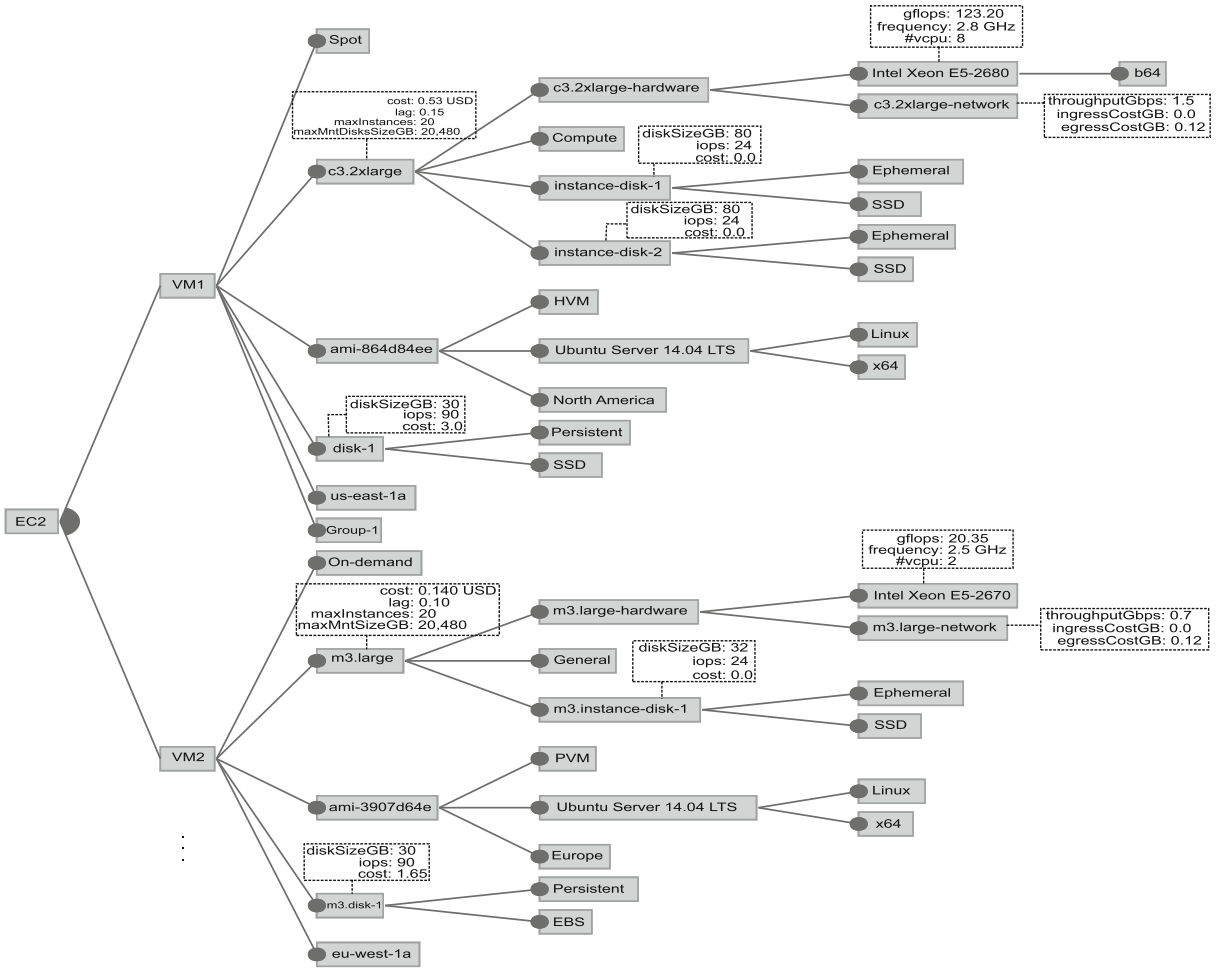
4.4 Configuration knowledge (CK)

As a feature may have many constraints, handling it manually is usually time-consuming and error-prone. To tackle these issues, the configuration knowledge (CK) contains the steps defining how each feature is instantiated, its requirements, and post-conditions. For instance, if an *instance type* demands *hardware-assisted* virtualization, an appropriate image is selected. Moreover, if it offers *instance disks*, its disks are mounted and formatted after it has been launched.

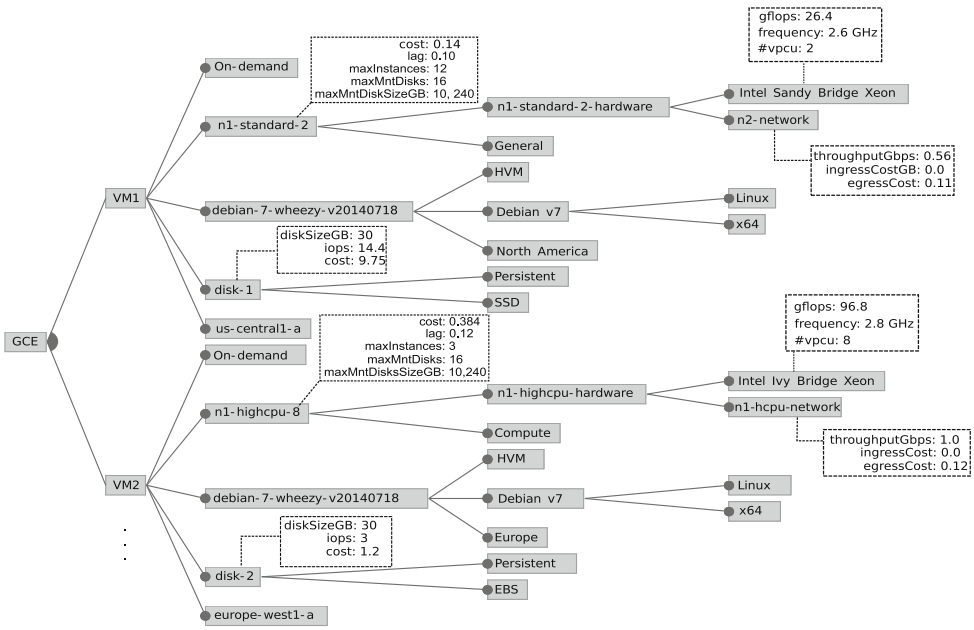
In practice, this means that the CK makes a mapping between the features and the artifacts implementing them. For instance, when a VM is started, it must be contextualized, as depicted in Fig. 5 to: (i) communicate with the other virtual machines running on the same cloud and on the other clouds; (ii) have the required software packages and security rules up-to-date; and (iii) allow the users to connect to any VM, without having to manually importing their public keys (e.g., SSH keys) in each cloud and VM.

Additionally, the artifacts might also have *variation points*. Some of these variation points can be runtime information. Example of variation points includes hosts' name, IP addresses, classes' names, among others that must be provided by the system at runtime. They enable artifacts reuse and allow the *system engineers* to delegate applications' dependencies to be resolved at runtime. For example, if an application has to be deployed on a node and its database on another one, the information about the database's address must be recovered when configuring the application. Listing 1 illustrates one configuration script with three variation points. A *variation point* is defined between ${[and]}. In this example, the script is exporting a node's region name, its endpoint, and its zone's name.

[1] UnixBench is a test suite for Linux systems to analyze their performance with regard to CPU, I/O, system call, among other operations. Based on the result of each test, an index score value is calculated, i.e., the UnixBench computes the index score value of a system using the SPARCstation 20 as baseline.



(a) Amazon EC2



(b) Google Compute Engine (GCE)

Fig. 4 Concrete models describing the configuration options of EC2 and GCE

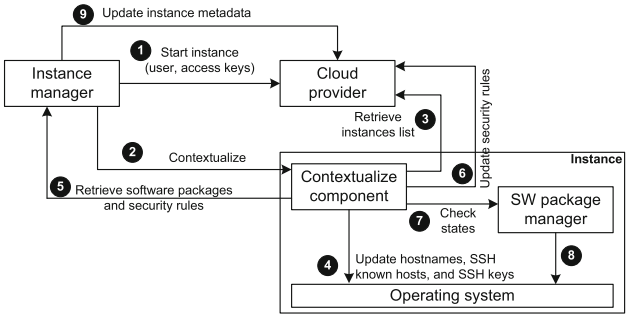


Fig. 5 The contextualization process to allow the virtual machine to share, for instance, the same access keys

```
1 ...
2  export NODE_REGION_NAME=${[location.region.name]}
3  export NODE_REGION_ENDPOINT=${[location.region.
       endpoint]}
4  export NODE_ZONE_NAME=${[location.name]}
5 ...
```

Listing 1 Example of one script with three variation points

We highlight that our approach still demands systems and domain experts to build the feature models (Sect. 4.3); i.e., our solution is not fully automated. However, this work is done only once, and it has a cost amortized along the time through the reuse of the models to instantiate the products many times. Nonetheless, information retrieval and natural language processing techniques can be used to implement automatic or semi-automatic feature model creation based on textual or semi-structured configuration descriptions [26].

5 Dohko

This section describes *Dohko*. *Dohko* is an autonomic system to help the users on dealing with inter-cloud environments. Particularly, *Dohko* is responsible for deriving a *deployment plan* to instantiate the computing environments on the clouds, executing the actions described in the CK.

Similarly, *Dohko* helps the users to rapidly know if their configuration requirements are available, freeing them from the work of reading extensive clouds' documentations. This is possible since *Dohko* uses a CSP solver to find valid configurations.

Section 5.1 describes *Dohko*'s architecture and its main components. Next, Sect. 5.2 presents the autonomic properties implemented by *Dohko*. Then, Sect. 5.3 illustrates how *Dohko* enables the users to execute a cloud-unaware application on the cloud following a declarative strategy.

5.1 Architecture overview

We assume an inter-cloud environment, where each cloud has a public application programming interface (API) that implements the operations to: (a) manage the virtual machines (e.g., to create, to start, to shutdown) and their disks; (b) get the list of the available VMs and their state (e.g., running, stopped, terminated, among others); (c) complement the VMs' descriptions through the usage of metadata. We also consider that the virtual machines fail following the fail-stop model. In other words, when a VM fails, its state changes to stopped and the system can either restart it or replace it by a new one.

As *Dohko* relies on a P2P overlay to organize the nodes, we assume that there is a set of nodes responsible for executing some specialized functions such as system bootstrapping and communication management. It is important to notice that the presence of such nodes does not mean a centralized control [3]. In addition, the number of these specialized nodes may vary according to some objective, e.g., the number of clouds.

Dohko uses a message queue system (MQS) to implement a publish/subscribe policy. The publish/subscribe interaction pattern is employed due to its properties such as [19]: space decoupling, time decoupling, and synchronization decoupling. These properties help the system to increase scalability and to make its communication infrastructure ready to work on distributed environments. Moreover, message queue systems normally do not impose any constraint for the infrastructure. In this case, all messages are persisted to support both node's and services' failures; and a message continues in the queue until a subscriber consumes it or until a defined timeout is achieved.

Dohko's architecture comprises three layers: *client*, *core*, and *infrastructure* as depicted in Fig. 6. There is also a cross-layer called *monitoring*. In the context of this work, the rationale for using a layered architecture is that it enables a loosely coupled design and a service-oriented architecture (SOA) strategy.

5.1.1 Client layer

The *client* layer provides the *job submission* module. This module takes an *application descriptor* as input, and submits it to a node in the cloud. This node is called *application manager*, and it will coordinate the execution of the applications across the clouds. Every node can be an *application manager*. In other words, an *application manager* is a temporary role of a node.

An *application descriptor* has five parts (Listing 2): user, requirements, clouds, applications, and on-finished action. The **user section** contains the user's information such as user name and his/her access keys. If the user does not have the access keys, they will be generated by the system. These keys are used to create and to connect to the virtual machines. The **requirements section**, on the other hand, includes: (a) the maximal cost to pay for a VM per hour; (b) the mini-

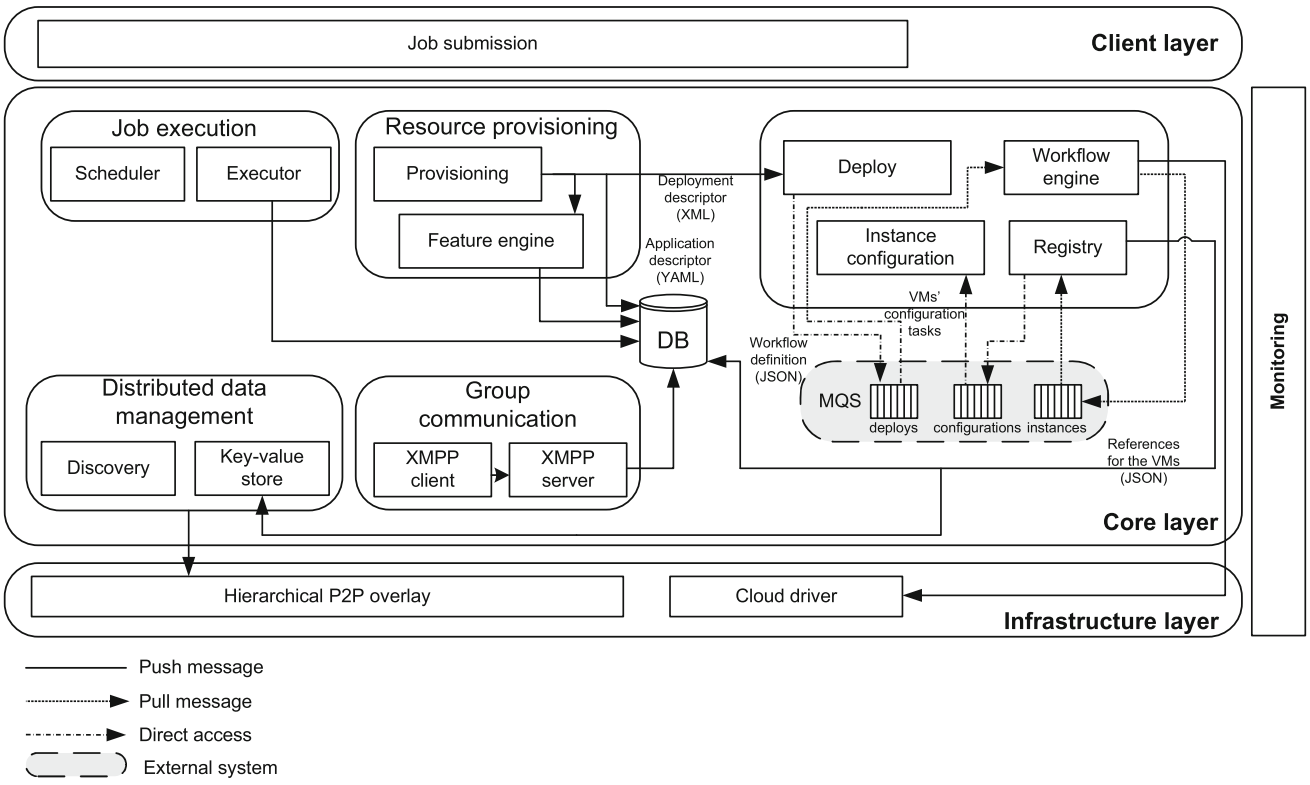


Fig. 6 *Dohko*'s architecture

mal number of CPU cores and RAM memory size; (c) the operating system; and (d) the number of virtual machines to be created in each cloud. These requirements are used to select the instance types based on the software product line engineering (SPLE) method (Sect. 4). The **clouds section** comprises the data of the user in each cloud provider. These data consist of the access and the secret key required to invoke the clouds' API, and they are given to users by the cloud providers. This section also contains informations such as region and instance types. However, these parameters target advanced users (e.g., system administrators) who may already have the regions and/or the instance types to be used. The **applications section** describes the tasks to be executed including their inputs and outputs. Finally, the **on-finished section** (Listing 42 in Listing 2) instructs the system about what to do after the applications have finished. The options are: *NONE*, *FINISH*, and *TERMINATE*. The *FINISH* option shuts down the virtual machines, which means that their persistent disks continue in the clouds; whereas the *TERMINATE* one shuts down and deletes the virtual machines.

```
1 ---
2 name:

4 user:
5   username:
6   keys:
7     key: []
```

```
9 requirements:
10   cpu:
11   memory:
12   platform:
13   cost:
14   number-of-instances-per-cloud:

16 clouds:
17   cloud:
18   - name:
19     provider:
20       name:
21     access-key:
22       access-key:
23       secret-key:
24     region:
25     - name:
26       zone:
27       - name:
28     instance-types:
29       instance-type:
30       - name:
31     number-of-instances:

33 applications:
34   application:
35     name:
36     command-line:
37     file:
38     - name:
39       path:
40       generated:

42 on-finished:
```

Listing 2 Structure of an application descriptor

5.1.2 Core layer

The *core* layer of the architecture (Fig. 6) comprises the modules to provision, to create, to configure, to implement group communication, and to manage data distribution, as well as to execute the applications.

The *provisioning* module is responsible for receiving an *application descriptor* and for generating a *deployment descriptor*. It utilizes the *feature engine* to obtain the instance types that meet users' requirements. The *feature model* implements the feature engine (Sect. 4.2) [40]. The *deployment descriptor* comprises all data required to create a virtual machine: (a) the cloud provider; (b) the instance type; (c) the zone (data center); (d) the virtual machine image; (e) the network security group and its inbound and outbound traffic rules; (f) the disks; and (g) the metadata. Listing 3 shows an example of a *deployment descriptor*. In this example, one virtual machine must be created with the name *e8e4b9711d36f4fb9814fa49b74f1b724-1* in the zone *us-east-1a* of region *us-east-1*. The cloud provider must be Amazon, using the instance type *t2.micro* and the virtual machine image *ami-864d84ee*. Furthermore, two metadata (i.e., tags) are added to the virtual machine.

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment user="username">
    <uuid>f4070433-a551-4a60-ba57-3e771ceb145f</uuid>
    <node name="e8e4b9711d36f4fb9814fa49b74f1b724-1"
     count="1" region="us-east-1"
         zone="us-east-1a">
        <provider name="amazon">
            <image>ami-864d84ee</image>
            <instance-type>t2.micro</instance-type>
        </provider>
        <tags>
            <tag>
                <name>app-deployment-id</name>
                <value>
      e8e4b9711d36f4fb9814fa49b74f1b724</value>
            </tag>
            <tag>
                <name>manager</name>
                <value>i-a1f8798c</value>
            </tag>
        </tags>
    </node>
</deployment>
```

Listing 3 An example of a deployment descriptor generated by the provisioning module

The *deployment descriptor* is sent to the *deploy* module. Then, the *deploy* module creates a workflow (*deployment workflow*) with the tasks to instantiate the VMs. A workflow is used since there are some precedent steps that must be performed in order to guarantee the creation of the VMs. For example, (a) the SSH keys must be generated and imported into the clouds; (b) if the instance types support the *group* feature, the system must check if one exists in the given zone and if not, create it. Furthermore, using a workflow enables the system to support partial failures of the deployment process. In addition, it decouples the architecture from the clouds' drivers. Thus, based on the provider's name, the *deploy* module selects in the database its correspondent driver to be instantiated at runtime by the *workflow engine*. This *deployment workflow* is enqueued by the *deploy* module and dequeued by the *workflow engine*. The *workflow engine* executes the workflow and enqueues the references of the created virtual machines in another queue.

The *registry* module is responsible for: (a) storing the instances informations in a local database and in the distributed key-value store; and (b) selecting the configurations (i.e., software packages) to be applied in each instance. These configurations are usually defined by the *system engineers*, and they represent a set of configuration scripts (Sect. 4.4).

The configuration tasks are put in a queue to be executed by the *instance configuration* module. The *instance configuration* module connects to the virtual machine via SSH and executes all the scripts, installing the software packages. This process guarantees that all instances have the required software packages.

The *group communication* module uses the Extensible Messaging and Presence Protocol (XMPP) (*xmpp.org*) for instant communication among the nodes in a cloud. It has an *XMPP server* and an *XMPP client*. The architecture uses the XMPP since some other group communication techniques such as broadcast or multicast are often disabled by the cloud provider. In addition, it supports the *monitoring* layer through its presence states (e.g., available, off-line, busy, among others).

The *job execution* module comprises a *scheduler* and an *executor*. They are responsible for implementing a task allocation policy and for executing the tasks. The task allocation policy determines in which node a task will be executed, considering that there are no temporal precedence relations among the tasks [8]. The goal of a task allocation/scheduling problem may be: (a) minimize the execution time of each task; (b) minimize the execution time of the whole application; (c) maximize the throughput (number of tasks completed per period of time). On its generic formulation, this problem has been proved NP-complete [24]. For this reason, several heuristics have been proposed to solve it. By default, *Dohko* provides an implementation of the self-scheduling (SS) [71] task allocation policy. However, other task allocation policies can be implemented and added to our system; i.e., our system does not assume any specific task allocation policy.

The self-scheduling (SS) policy assumes a master/slave organization, where there is a master node responsible for allocating the tasks, and several slaves nodes that execute the tasks. Moreover, it considers that limited information is available about the execution time of the tasks and the computing power of the nodes. In this case, it distributes the tasks, one by one, as they are required by the slave nodes. Thus,

Table 1 Main operations implemented by the key-value store

Operation	Description
void **insert** (key, value)	inserts the value into the network with the given key. If two or more values exist with the same key, all of them are stored
Set<Value> **retrieve** (key)	returns all the values with the given key
remove (key)	removes all the values stored under the given key
void **remove** (key, value)	removes the value stored under the given key

each node always receives one task, executes it, and, when the execution finishes, asks for one more task [71].

In our system, the node that receives the *application descriptor* becomes the *application manager*, and it is the responsible for creating and coordinating the configuration of the other nodes, as well as to distribute the tasks to be executed by them.

The *distributed data management* module provides a *resource discovery* and a distributed *key-value store* service. These services are based on a *hierarchical P2P* [68,79] overlay available at the *infrastructure* layer (Sect. 5.1.3). Table 1 shows the operations implemented by the *key-value store*. Basically, it providers three operations: (i) insert, (ii) retrieve, and (iii) remove. The insert operation takes a key and a value, and stores the value under the given key. In case there exists a value with the same key, all of them they are stored. In other words, the value of a key may be a set of objects. The retrieve method receives a key and returns its values. Finally, the remove method can remove all values of a given key or only one specific value.

5.1.3 Infrastructure layer

The *infrastructure* layer consists of the *hierarchical P2P* overlay and the *cloud drivers*. The *hierarchical P2P* overlay is used to connect the clouds and their nodes. In this case, in a cloud with n nodes, where $n > 0$, $n - 1$ nodes (i.e., leaf-nodes) join an internal overlay network and one node (i.e., super-peer) joins the external overlay network. In other words, there is one overlay network connecting the clouds and another overlay network in each cloud connecting its nodes. The super-peer and its leaf-nodes communicate through a HTTP service, and the leaf-nodes monitor the super-peer via XMPP. Both overlays are implemented using the Chord [67] protocol. Figure 7 illustrates the hierarchical P2P overlays connecting two clouds, each one with four nodes.

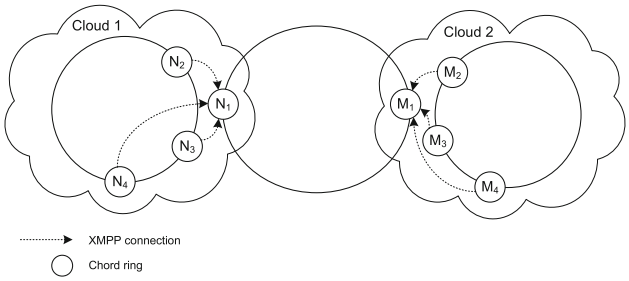


Fig. 7 Structure of the hierarchical P2P overlay connecting two clouds

When a node n running in cloud c starts, it asks the bootstrapping node, through a HTTP service, the super-peer of cloud c. If node n is the first peer of cloud c, it joins the super-peers overlay by the bootstrapping node. Otherwise, queries the super-peer for its leaf-nodes and joins the leaf-nodes overlay network or creates a overlay network. After it has joined one overlay network, the node stores in the key-value store its information under the keys: $/c/n$, if it is the super-peer or $/c/ <$ super-peer's id $>$ /members, otherwise.

Leaf-nodes of different clouds communicate through their super-peers. In this case, when a leaf-node wants to communicate with a node outside its cloud, it sends a message for its super-peer that first connects to the super-peer of such node, and next forwards the message for it.

When a node leaves one cloud, it notifies its super-peer that removes the information about the node from the system.

5.1.4 Monitoring cross-layer

The *monitoring* layer is responsible for (a) checking if there are virtual machines that were not configured; (b) detecting and restarting failed nodes; (c) keeping up-to-date the information about the super-peers and the leaf-nodes in the system.

5.2 Autonomic properties

This section presents how our system implements the following autonomic properties: self-configuration, self-healing, and context-awareness. Although there exists other autonomic properties, as described in [29,35], our system implements only these properties since our focus is to help the users on tackling the difficulties of deploying and executing applications across multiple clouds, taking into account different objectives, and without requiring cloud and system administration skills from the users. Figure 8 shows the autonomic control loop implemented by this work.

The process follows a declarative strategy. A declarative strategy allows the users to concentrate on their objectives rather than on dealing with cloud or system administration issues. In this case, the process starts with the users

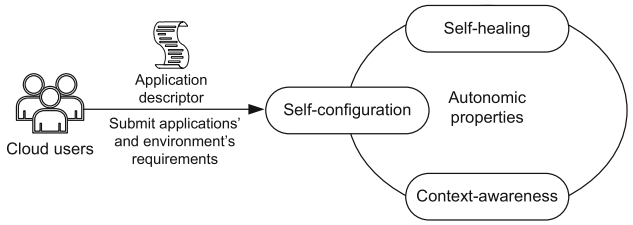


Fig. 8 The autonomic properties implemented by our architecture

describing their applications and constraints. Then, using a self-configuration process, the system (a) creates and configures the whole computing environment taking into account the characteristics and the state of the environment, i.e., the availability of other virtual machines; (b) monitors the availability and state of the nodes through the self-healing; (c) connects the nodes taking into account their location. In other words, using a hierarchical organization, nodes in the same cloud join an internal overlay network and one of them joins an external overlay network, connecting the clouds (Fig. 7). Nodes in the internal overlay network use internal IP addresses for communication, which often has zero cost and a network throughput higher than if they were using external IP addresses; finally (d) executes the applications.

In the next sections, we will explain each one of these properties in detail.

5.2.1 Self-configuration

Our system automatically creates and configures the virtual machines in the clouds. The configuration process is based on the *concrete feature* model [40] defined by the *system engineers* (Fig. 2) and on the users requirements. When the system receives an *application descriptor*, it (a) creates the access keys (i.e., the private and the public keys) to be used by the virtual machines, and imports them into the clouds; (b) creates a security group with the inbound and outbound rules; (c) uses the clouds' metadata support to describe the VMs, allowing the users to trace the origin of each resource, and to access them without using our system, if necessary. In addition, these metadata provide support for the self-healing process; (d) selects one availability zone to deploy the virtual machines according to the availability of other VMs running in the same zone; (e) selects the instance types; (f) configures the instances with all the software packages; (g) starts all the required services considering the instance state. For example, if a service requires one information (e.g., its region's name, up-to-date IP addresses) about the environment where it is running, it is automatically assigned by the our system before it starts. For instance, when an instance fails or needs to be restarted, the architecture detects the new values of its ephemeral properties such as the internal and external addresses, and starts its services to use up-to-date informations.

The metadata of an instance include: (i) the user name to access the VM, (ii) the value of the keys, (iii) the name of its virtual machine image (VMI), (iv) the owner (i.e., the user), (v) the name of its application manager, and (vi) the name of the feature model used to configure it. These data help the system to support failures of both the manager and the architecture. For example, suppose that just after having created the virtual machines, the application manager node fails. Without these metadata, another node could not access the instances to configure them, leaving for the system only the option to terminate the instances, which implies a financial cost as the users will pay for the instances without having used them.

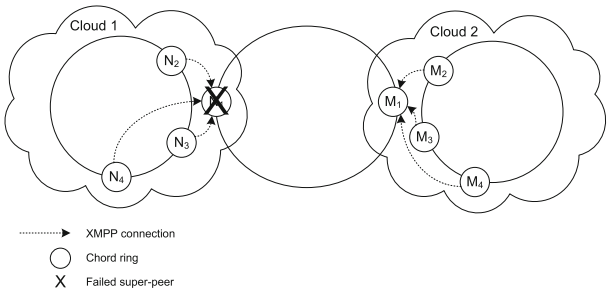
5.2.2 Self-healing

The cross-layer *monitor* module uses the XMPP to monitor the availability of the virtual machines. In this case, every virtual machine runs an XMPP server, which periodically requests the *discovery* service to list the VMs running in the cloud. The returned VMs are included as the monitor contact. In this context, each VM utilizes the XMPP's status to report to others its status and also to know their status. When a VM's status changes to off-line, its application manager waits a determined time and requests the cloud provider to restart it. The restarting avoids unresponsive node due to its workload state (i.e., overloaded VM) or some network issues. If the node's status does not change to on-line, the manager terminates it, and creates a new one. In case of the super-peer's failures (Fig. 9a), the leaf-node with the higher uptime tries to recover the failed super-peer. If it is impossible, this leaf-node leaves its network (Fig. 9b) and joins the super-peers network, becoming the new super-peer.

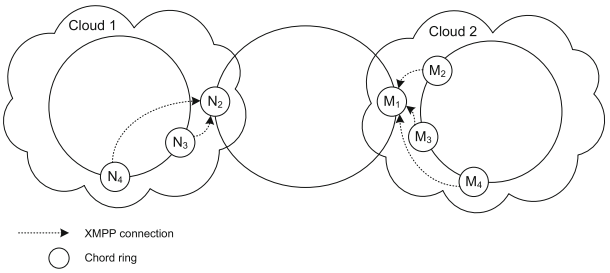
Since the application manager may completely fail during the configuration of the virtual machines, the *monitor* checks if there are instances created by the system that have not been been completely configured. If such instances do exist, it contacts their application manager or sends them for other node to continue their configuration.

5.2.3 Context-awareness

Similar to [44] and [4], by context we refer to the topology of the overlay network or peers association, which may impact the overall system's performance (e.g., throughput and latency), as well as in the financial cost (e.g., network communication cost). In other words, in this work, context-awareness means the capacity of the P2P communication protocol be aware of the peers' locations and of adapting the system's behavior based on the situation changes [44,76].



(a) Cloud's 1 super-peer (N_1) failed disconnecting the clouds 1 and 2



(b) Leaf-node N_2 leaves its overlay network and joins the new super-peer overlay network, connecting the clouds 1 and 2

Fig. 9 Example of super-peer failure and definition of a new super-peer

As described earlier in this section, the nodes are organized into two overlay networks. This avoids unnecessary data transfers between the clouds and often increases the network throughput between the nodes in the internal overlay. Furthermore, it helps to decrease the cost of the application execution, avoiding inter-cloud data transferring due to the overlay stabilization activities [67]. If all nodes were organized in the same P2P overlay network, they would have to communicate using external IP addresses (i.e., public IP), which implies in Internet traffic cost, even if the nodes are located in the same data center. In addition, in a high churn rate scenario [36], it decreases the cost of maintaining the distributed hash tables (DHTs) up-to-date since it does not require inter-cloud communication.

5.3 Using *Dohko* to execute a cloud-unaware application

In order to illustrate the usage of our system, consider that one user is interested in executing his/her application in the cloud (Fig. 10). In this example, the application manager is the node that receives the user's demand (*application descriptor*), and the worker is a node that receives a task to execute.

The process starts when the user defines an *application descriptor* depicted in Listing 4, with the requirements and applications. In this example, one virtual machine with at least 1 CPU core and 1 GB of RAM memory is requested, with the Linux operating system, and a cost of at most 0.02 USD/hour. Moreover, this instance has to be created on Amazon using the given access and secret key. Finally, the task consists of getting the information about the CPU (Listing 25 in Listing 4).

```
---
name: "example"
user:
  username: "user"
requirements:
  cpu: 1
  memory: 1
  platform: "LINUX"
  cost: 0.02
  number-of-instances-per-cloud: 1
clouds:
  cloud:
  - name: "ec2"
    provider:
      name: "amazon"
    access-key:
      access-key: "65AA31A0E92741A2"
      secret-key: "619770ECE1D5492886D80B44E3AA2970"
    region: []
    instance-types:
      instance-type: []
applications:
  application:
    name: "cpuinfo"
    command-line: "cat /proc/cpuinfo"
on-finished: "NONE"
```

Listing 4 Application descriptor with the requirements and one application to be executed in one cloud

The *application descriptor* is submitted to the system through the *job submission* module (Fig. 10(1)). Then, the *job submission* module looks for an appropriate node through the *discovery* service (Fig. 10(2)), and sends the *application descriptor* to it (Fig. 10(3)). Thus, this node becomes the *application manager* of the submitted application.

Afterwards, the *provisioning* module in the application manager takes the *application descriptor* and persists into its database (Fig. 10(4)). Next, the *provisioning* module obtains from the *feature engine* module: (a) an instance type that meets the user's constraints, and (b) a virtual machine image (VMI). The *feature engine* returns the *t2.micro* instance type in the *us-east-1* region, and the virtual machine image *ami-864d84ee*. With these data, the *provisioning* module: (a) selects a zone to host the virtual machine, (b) generates the *deployment descriptor* (Listing 3), and (c) submits it to the *deployment* module (Fig. 10(5)).

The *deployment* module creates a workflow (Fig. 11) with the steps to instantiate the VM, and enqueues it into the MQS (Fig. 10(6), *deploys* queue). The *workflow engine* executes the deployment workflow, i.e., it connects to the cloud and creates the virtual machine (Fig. 10(7 and 8)). The data about the VM are enqueued in the *instances* queue (Fig. 10(9)). Next, the *registry* module dequeues the instance from the queue *instances* and inserts its information into the database and into the *key-value store* (Fig. 10(10)). It then creates the configuration tasks to be executed in the

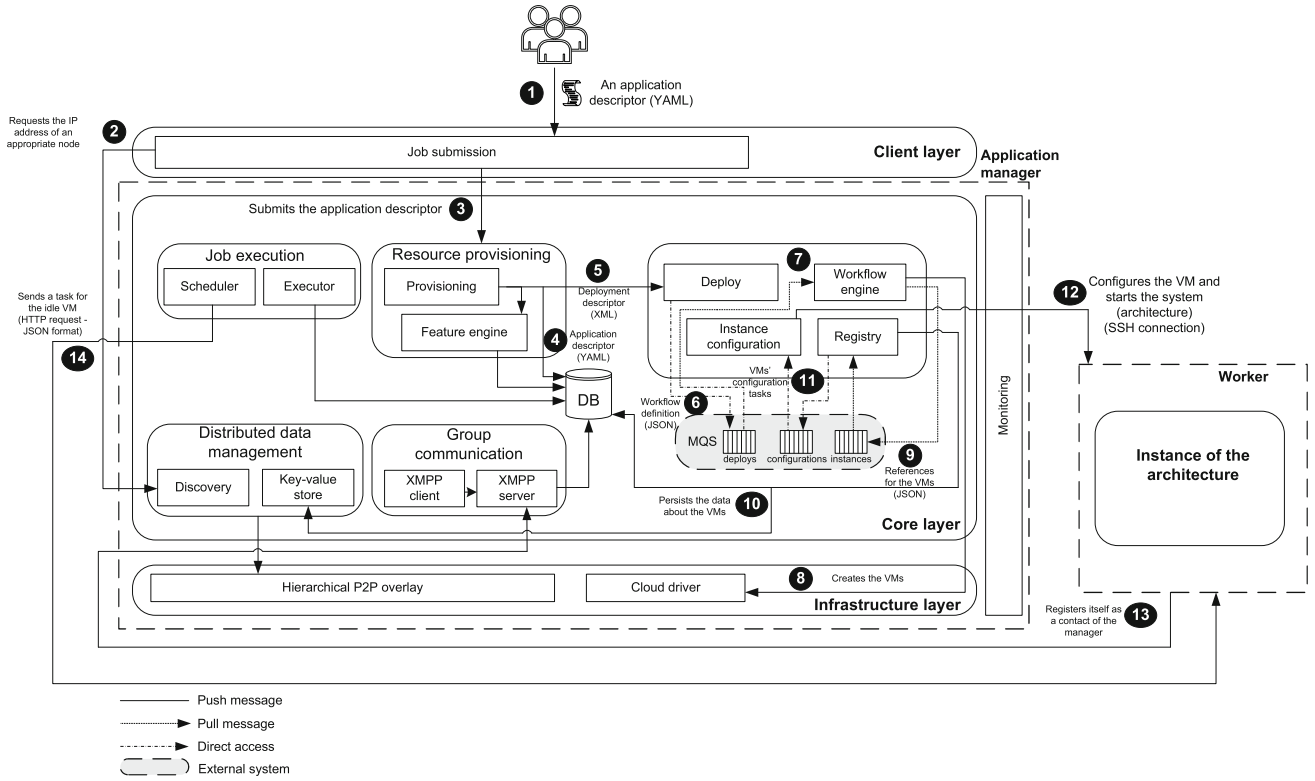


Fig. 10 Interaction between *Dohko*'s modules when an application descriptor has been submitted for execution

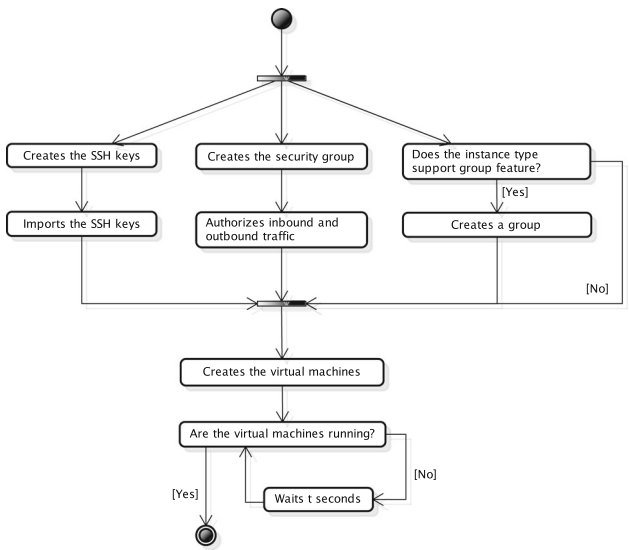


Fig. 11 Workflow to create one virtual machine in the cloud

virtual machine (Fig. 10(11)). Each task comprises a host, a user name, the SSH keys, and the scripts to be executed. Subsequently, the *instance configuration* module: (a) connects to the node via SSH; (b) executes the scripts; and (c) starts an instance of *Dohko* (Fig. 10(12)). Then, the virtual machine, which is now executing a Dohko's instance, notifies the application manager using the *group communication* module (Fig. 10(13)). Finally, the manager uses the *scheduler* (Fig. 10(14)) to distribute the tasks, according to the self-scheduling (SS) task allocation policy (i.e., self-scheduling (SS) [71]). In other words, the *scheduler* sends a task to the worker, which should execute it and return its result to the application manager. The whole process is monitored by the *monitoring* module.

5.4 Implementation

We entirely modeled the clouds of Amazon and Google Compute Engine (GCE). Both *Dohko* and the model were implemented in Java, and their source code is available at *dohko.io*. Our implementation uses Choco [33] as the constraint satisfaction problem (CSP) solver. Choco [33] was used since it is a well-known satisfaction problem solver, and also because it has been successfully used elsewhere with similar purpose [18,69].

6 Evaluation

We evaluated our proposed approach considering single and inter-cloud scenarios. In this case, the goal was: (i) to analyze the complexity of our process; (ii) if it allows the users to execute their applications across multiple clouds infrastructures,

without needing to deal with the specificity of each cloud provider; (iii) if it enables a declarative strategy; (iv) if our approach can deal with resource failure taking into account resource location following a distributed architecture.

6.1 Goal question metric

The evaluation was organized with the goal question metric (GQM) [5]. The questions and the correspondent metrics were defined as follows:

Question 1: *Can our solution automate resource selection in an inter-cloud scenario basing only on the data known by ordinary users such as the minimal number of CPU cores, RAM memory size, and the maximum financial cost per hour?*

Metrics

M.1.1 number of selected instance types that offer the best cost per feature

M.1.2 number of selected instance types that without specific cloud computing skills may be a difficult choice for unskilled users

Question 2: *Can our solution enable a homogeneous inter-cloud management, without being tied to any provider?*

Metrics

M.2 number of completed demands for resource allocation on the providers defined by the users, employing our proposed inter-cloud management interface

Question 3: *Can our solution completely automate inter-cloud configuration, following a declarative description?*

Metrics

M.3.1: number of times that the applications could be deployed across the clouds using our application descriptor

M.3.1: the average time to setup a computing environment on the clouds

Question 4: *Can our solution deal with resource failure without demanding a centralized architecture?*

Metrics

M4.1: the average time demanded by our system to detect and to recover from resources' failures on the cloud

M4.2: the average time that different resource's failures has in the overall application execution

6.2 Environment setup

Dohko uses RabbitMQ as the MQS and the Linux distributions Debian and Ubuntu in the nodes in the clouds. In this case, Debian was used in the experiments executed in GCE and Ubuntu in the ones executed in EC2. Table 2 presents the setup of the experimental environment.

Table 2 Setup of the experimental environment

Software	Version
GNU/Linux x86_64	3.2.0-4-amd64
GNU/Linux x86_64	3.13.0-29-generic
OpenJDK	1.7.0_65
RabbitMQ	3.3.5
SSEARCH	36.3.6

Table 3 Users' requirements without a preference to a provider

# Req.	# vCPU	Memory (GB)	Cost (USD/h)
1	4	4	0.5
2	16	8	1.0
3	16	90	2.0

6.3 Simulation setup

In order to evaluate our solution, we simulated the resources requirements depicted in Table 3. In this case, we assumed that the users want to minimize the monetary cost (M1.1 and M1.2) and to maximize resource capacity. Hence, in Table 3 the requirements are defined as the minimal number of vCPUs and the amount of memory; and the maximal financial cost per hour (Question 1). These requirements were defined based on the amount of resources normally available on ordinary users' computers, and on the resources demanded by the application.

Likewise, we used the self-scheduling (SS) task allocation policy [71] to execute the *SSEARCH* as a parameter sweep application. A parameter sweep application is defined as a set $T = \{t_1, t_2, \ldots, t_m\}$ of m independent tasks. In this context, independence means that there is neither communication nor temporal precedence relations among the tasks. Besides that, all the m tasks execute exactly the same program changing only the input of each task [7]. In this work, a task comprises an execution of the SSEARCH program to compare a genomics query sequence with UniProtKB/Swiss-Prot database.

The *SSEARCH* (*www.ebi.ac.uk/Tools/sss*) is a biological comparison program, which performs the Smith- Waterman (SW) [64] algorithm. The Smith-Waterman (SW) algorithm [64] is an exact algorithm based on dynamic programming to obtain the best local alignment between two sequences in quadratic time and space. Particularly, it may require petabytes of memory to store its dynamic programming matrices when comparing long biological sequences,

Table 4 Users' requirements to deploy the SSEARCH application on the clouds

# Req.	# vCPU	Memory (GB)	Cost⋆	# VM	CP[b]
	2	6	0.2	5	
1	2	6	0.2	10	EC2
	2	6	0.2	5	
2	2	6	0.2	10	GCE
	4	6	1.0	5	
3	4	6	1.0	10	EC2
	4	6	1.0	5	
4	4	6	1.0	10	GCE
	4	6	1.0	5	EC2
5	4	6	1.0	10	GCE

⋆ USD/h
[b] Cloud provider

and it may demand days or even weeks to compute the alignment. It is commonly used by biologists to identify functional, structural, or evolutionary relationships between biological sequences.

In our tests, the SSEARCH application compared 24 sequences with the database UniProtKB/Swiss-Prot (September 2014), available at *uniprot.org/downloads*, composed of 546, 238 sequences. The query sequences are the same presented in [39]. For each sequence, we defined an entry in the *application descriptor*, i.e., each sequence represents a task.

We also simulated a scenario where the users defined the providers and the number of virtual machines to create and to configure (M2) on the clouds, using a deployment descriptor (M3.1). Table 4 presents the scenarios and the constraints, and Listing 5 exemplifies an *application descriptor*. In this example, we requested at least 2 cores and 6 GB of RAM memory, the Linux operating system, and a cost of at most 0.2 USD/hour. Moreover, we asked to execute the *ssearch36* application. The *ssearch36* receives as input the query sequences and the UniProtKB/Swiss-Prot database; and its output is a score table.

Next, we measured the deployment time, i.e., the time to create and to configure the virtual machines on the clouds (M3.2). The wallclock time was measured including: (a) the time to instantiate the VMs on the clouds; (b) the time to download and to configure all the software packages; and (c) the time to start *Dohko* on each node.

Furthermore, we evaluate the self-healing property of our system to handle failures of clouds' resources (M4.1) employing a decentralized strategy. Finally, we checked the impact on the execution time when the application manager, the worker nodes or both fail (M4.2).

Each experiment was repeated three times, and the mean was taken.

```
---
name: "ssearch-app"
user:
  key: []
  username: "user"
requirements:
  cpu: 2
  memory: 6
  platform: "LINUX"
  cost: 0.2
clouds:
  cloud: []
applications:
  application:
    name: "ssearch36"
    command-line: "ssearch36 -d 0 ${query} ${database
        } >> ${score_table}"
    file:
    - name: "query"
      path: "$HOME/sequences/O60341.fasta"
      generated: "N"
    - name: "database"
      path: "$HOME/uniprot_sprot.fasta"
      generated: "N"
    - name: "score_table"
      path: "$HOME/scores/O60341_scores.txt"
      generated: "Y"
  ...
on-finished: "TERMINATE"
```

Listing 5 An application descriptor with one SSEARCH description to be executed in the cloud

6.4 Result and analysis

6.4.1 Resource selection

Considering resource selection, our approach could select the instances on the clouds, taking into account the users constraints. Particularly, it could enable ordinary users to have access for resource types that demand advanced skills. For example, there are two instance types that meet the # Req. 3 described in Table 3. The candidates are the *n1-highmem-16* and the *r3.4xlarge* (Table 5). Although the former costs 6% less than the latter, our system selected the latter, since it offers 17% more RAM memory; has a sustainable performance (GFlops) higher than the delivered by the other one and an SSD instance disk of 320 GB with zero cost. Furthermore, it has a network throughput of 9.3 Gbps. Clearly, it may be difficult for the users deciding to use the *r3.4xlarge*, taking into account that they want lower cost and not all these informations are available on the documentations.

6.4.2 Application deployment

This experiment aims to measure the deployment time, i.e., the time to create and to configure the virtual machines. The wall-clock time was measured including: (a) the time to instantiate the VMs in the cloud provider; (b) the time to download and to configure all the software packages (e.g., Java, RabbitMQ, SSEARCH, among others); and (c) the time to start the architecture. By default, the architecture performs

Table 5 Instance types that has at least 16 CPU cores, 90GB of RAM memory, and a cost less than 2 USD/h

Instance type	# vCPUs	RAM (GB)	Cost (USD/H)	GFlops	Network	Cloud provider
n1-highmem-16	16	104	1.312	79.06	0.8	GCE
r3.4xlarge	16	122	1.400	138.86	9.3	EC2

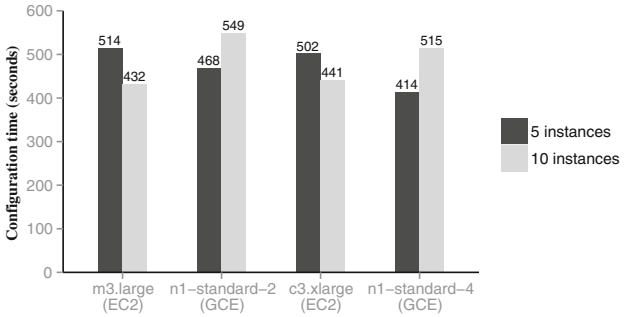


Fig. 12 Configuration time of the virtual machines on the clouds

the configurations in parallel, with the number of parallel processes defined in a parameter of the architecture. In this case, the maximum of 10 configurations were done in parallel.

Figure 12 presents the deployment time for the instance types listed in Table 6. As can be seen, in Amazon, increasing the number of virtual machines to deploy from 5 to 10 decreases the deployment time. This occurs because the virtual machines of each experiment are homogeneous, which enables us to request multiple instances at the same time. Similar behavior has already been observed by other works in the literature [31,48]. On the other hand, in Google, the deployment time is proportional to the number of virtual machines requested. This overhead might occurs since for each virtual machine the following steps must be executed: (a) select a physical machine to host the VM; (b) create the disks to the new VM; (c) copy the virtual machine image to the select host; (d) request the hypervisor to start the virtual machine; (e) configure an IP address, imports user's access keys, among other configurations; (f) test the state of the VM; and (g) start to charge for its usage.

In our experiment, the deployment time of 5/10 VMs took at most 10 minutes. With this, the applications can start across multiple VMs and multiple clouds, without requiring from users cloud and system administration skills, and also without needing the use of virtual machine image (VMI).

For the # Req. 4 in Table 4, our system allocated 5 VMs in U.S. and 5 in Europe, since GCE imposes a limit of 6 virtual machines of this type in each region. This constraint was handled by the *concrete feature models* and the *configuration knowledge*.

6.4.3 Application execution

Figure 13 presents the wall-clock execution time for the four standalone experiments (i.e., requirements 1 to 4 of Table 4), and in Table 7, we have their total financial cost. We can see that the instances that belong to the same family type have almost the same performance. The lower execution time (89 seconds) was achieved by the instance type *c3.xlarge* (40 vCPU cores) with a cost of 2.10 USD. Considering that the application does not take a long time to finish neither it demands many computing resources, this represents a high cost. If the users wait 33% more (43 seconds), they can pay 50% less (1.05 USD).

In Fig. 14, we present the execution time for a multi-cloud scenario. In this case, for the requirement of 5 instances (i.e., # Req. 5 of Table 4), 3 virtual machines were used from EC2 and 2 instances from GCE. If we compare Fig. 13 with Fig. 14, we can see that the execution time increased almost 34%. One reason for this difference is probably due to the network throughput between the clouds of different providers, since we do not observe such overhead in the scenario with ten *n1-standard-4* virtual machines distributed across two GCE clouds.

Figure 15 presents the total time (i.e., deployment + execution time) for the experiments. In this figure, we can observe the impact of deployment time of 10 virtual machines has in the total execution time in GCE environment. In this case,

Table 6 Instance types automatically selected by our system based on the users requirements

# Req.	Instance type	# vCPU	RAM (GB)	Cost (USD/h)	Family type	Cloud provider
1	m3.large	2	7.5	0.14	General	EC2
2	n1-standard-2	2	7.5	0.14	Memory	GCE
3	c3.xlarge	4	7.5	0.21	Compute	EC2
4	n1-standard-4	4	15	0.28	General	GCE
	c3.xlarge	4	15	0.21*	General	EC2
5	n1-standard-4	4	15	0.28	General	GCE

* 3 *c3.xlarge* and 2 *n1-standard-4*

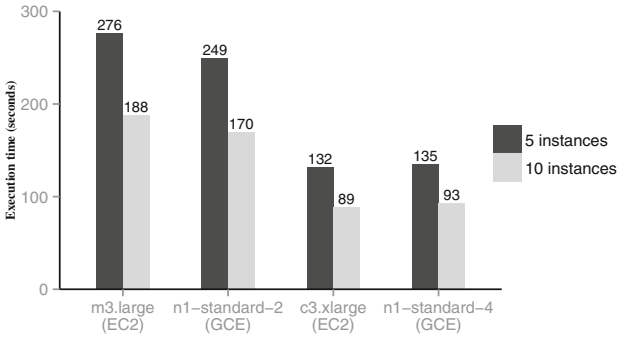


Fig. 13 SSEARCH's execution time on the clouds to compare 24 genomics query sequences with the UniProtKB/Swiss-Prot database

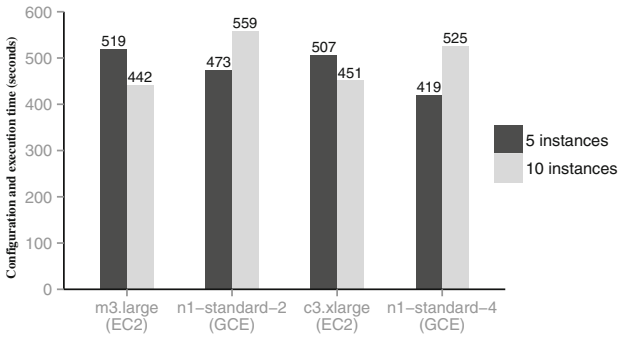


Fig. 15 Deployment and execution time of the experiments

Table 7 Financial cost for executing the application in the cloud considering different requirements (Table 4)

Instance type	# Instances	Wall-clock time[±]	Total cost (USD)
m3.large	5	276	0.7
m3.large	10	188	1.4
n1-standard-2	5	249	0.7
n1-standard-2	10	170	1.4
c3.large	5	132	1.05
c3.large	10	89	2.10
n1-standard-4	5	135	1.4
n1-standard-4	10	93	2.94[⋆]
c3.large	3		0.63
n1-standard-4	2	131	0.53[♯]
c3.large	5		1.05
n1-standard-4	5	119	1.40[♭]

[±] In seconds
[⋆] *Total cost: 2.96 (USD) (1.4 (U.S.) + 1.54 (Europe))*
[♯] Total cost: 1.16 USD
[♭] Total cost: 2.45 USD

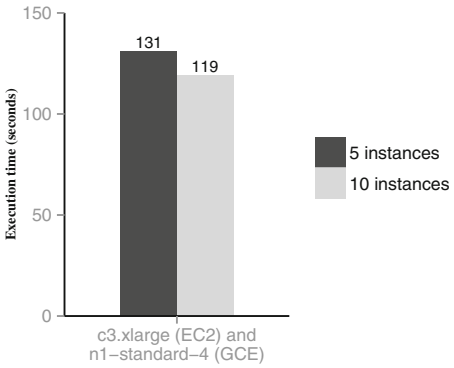


Fig. 14 Execution time of the SSEARCH application to compare 24 genomics query sequences with the UniProtKB/Swiss-Prot database in an inter-cloud scenario

the deployment time of 5 virtual machines was better than the deployment time of 10 virtual machines.

6.4.4 Application deployment and execution with failures

To evaluate the self-healing property of our architecture, we simulated two types of failures. In the first case, we stopped the application manager just after it had received the tasks to execute. In the second case, we executed a process that, at each minute, selected and stopped a worker. These failures were evaluated in an inter-cloud scenario (10 virtual machines – 5 *c3.xlarge* and 5 *n1-standard-4*).

Figure 16 presents the execution time of the application for each failure scenario. As can be seen, the failures of the workers significantly impact the total execution time. This mostly occurs because worker's failures demands a time from the application manager to detect it and to reassign the task of the failed node to another one. Moreover, since multiple workers failed, this highly impacted the total execution time compared with both the failure-free scenario and the failure of the application manager. Moreover, when the application manager fails, the workers can still continue the execution of their tasks.

Next, we simulated failures of the super-peers to evaluate both the resilience of our hierarchical P2P overlay network and if their failures impact the execution time. In this case, the work consisted in requesting the super-peer to leave the external overlay. Therefore, these type of failures do not increase the execution time since: (a) the overlays and the applications are independent processes; (b) the application consists of embarrassingly parallel tasks (i.e., bag-of-tasks (BoT)), and (c) our system relies on both Chord [67] and XMPP protocols to detect node leaving the overlay and to assign a new super-peer.

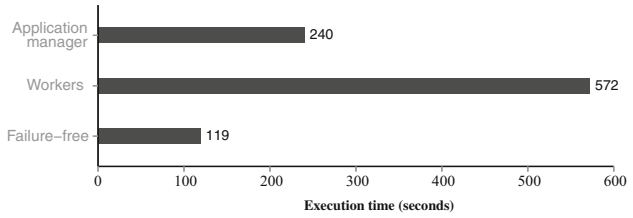


Fig. 16 Execution time of the application on the clouds with three different type of failures

6.5 Threats to validity

There are some concerns related to the validity of our contributions described in this work. First, our tool and the feature models were built based on our experience in developing, deploying, and configuring inter-cloud services, as well as in administrating computing environments. Second, the concrete feature models were limited to the features released by the cloud providers. Third, since our method relied on benchmarks to gather the qualitative attributed of the resources, small variations in their result values might occur mostly for two reasons: (i) we cannot control the allocation of the physical resources to the virtual ones and (ii) clouds' workload may change over time, as well as the cloud providers' objectives. Fourth, due to the evolution of cloud computing, some resources, clouds or even providers may appear and/or disappear. Thus, the experiments and concrete feature models presented in this work might be invalid over the long term. However, this does not invalidate our *abstract feature model* (Fig. 3), since it is independent of cloud provider. Moreover, our tool is not invalidated too, because its modules are decoupled from the feature models. Finally, although the examples deal with public cloud providers our contributions are not limited to these clouds, as other cloud platforms follow the same principles employed by these cloud providers.

7 Related work

Feature modeling has been used to handle the variability of VMI, multi-tenant applications, native cloud deployment, and clouds' service selection. In this case, Sect. 7.1 briefly reviews some of the works available in the literature.

Since the proposal of autonomic computing, many autonomic computing systems have been proposed in the literature. Thus, Sect. 7.2 reviews some of these autonomic systems designed for cloud environment.

7.1 Cloud configuration management

Cloud configuration management tools have been considered by some works in the literature. Some of these works [9,55] have focused on handling the variability of the platform-as-a-service (PaaS) layer. In this case, they aim to support the developers on deploying their application on the cloud or to support them on developing native cloud applications. Other works [18,69,70] have employed feature models to describe the configuration of virtual machine images with different objectives. For instance, to reduce: (a) energy consumption [18,69] and (b) storage space [78] used with pre-built virtual machine images.

Table 8 summarizes the papers available in the literature. The first column shows the paper or the name of the solution presented in the paper. Then, the second column presents how feature modeling is used to deal with clouds' variabilities. The cloud service model considered by each paper is described in the third column. The last column, on the other hand, describes the objectives of each paper.

Cloud service selection [75] and configuration of multi-tenant applications have been addressed by [58],[60], and [59]. These works focus on supporting the users on customizing SaaS applications. In [75], the users define multiple models of a service and submit them to a system that selects one that meets their objectives. Feature model has been used to model the variability of one specific IaaS provider [23].

Most of the works [11,18,38,70,78] handle the variabilities of virtual machine images. In this case, they aim to minimize the time to setup a virtual machine considering performance constraints and in some case, power consumption [18,69]. Feature modeling has also used to reduce storage space [78] used with pre-built images or to support the users on deploying a virtual machine in the cloud.

At the PaaS layer, one solution [2,9] uses feature modeling to help the developers on changing the code of an application to use cloud's services; and another solution [54,55] uses it to support the developers on deploying their native cloud applications.

Managing the variability of SaaS applications has be considered by multiple works [46,58–60,74] with different objectives. In such case, they aim to avoid invalid configurations, to reduce the time to configure or to create an application, and to avoid single tenant deploys.

Walraven and colleagues [74] use feature modeling to enable resource selection in a specific cloud. In this case, based on users' requirements, a feature model is employed to obtain the configurations of the cloud and to check that a user's request is valid.

Sousa and colleagues [65] have employed feature modeling for managing the variability of cloud providers to deploy microservices. A microservice is an architectural style to build applications based on lightweight and independent service that performs a single function and that collaborates with other services using a well-defined interface to achieve some objectives [43].

Table 8 Comparative view of cloud variability models

Paper	Usage of feature model	Cloud model	Inter-clouds	Objectives
SCORCH [18]	VMI configuration	IaaS	No	Minimize power consumption taking into account performance constraints
[69,70]	VMI configuration	IaaS	No	Minimize the time to setup a VM and power consumption
[78]	VMI configuration	IaaS	No	Minimize storage space with pre-built VMI and the time to setup a VM
[38]	VMI deployment	IaaS	No	Minimize the time to deploy a VM
[11]	VMI deployment	IaaS	No	Minimize the time to deploy an application
HW-CSPL [2,9]	Development of a cloud-aware application	PaaS	No	Minimize the time to change an application to use clouds' services
SALOON [54,55]	Deployment of native cloud applications	PaaS	No	Minimize the time/effort to deploy native cloud applications
[65]	Configuration and deployment of microservice-based applications	PaaS	Yes	Help the developers on deploying microservices on multiple clouds
[46]	CMMTA⋆	SaaS	No	Minimize the time to configure an application
[58,59]	CMMTA⋆	SaaS	No	Minimize the number of single-tenant deploys taking into account users' security constraints
[60]	CMMTA⋆	SaaS	No	Minimize inconsistent feature models
[74]	CMMTA⋆	SaaS	No	Minimize the time to create a cloud-aware application
[23]	Instance selection	IaaS	No	Minimize invalid configurations considering a single objective
This work	**IS and IC♯**	**IaaS**	**Yes**	**Avoid invalid configurations and handling service selection and configuration on multiple clouds, taking into account multiple objectives**

⋆ Configuration management of multi-tenant applications (CMMTA)
♯ Instance selection and infrastructure configuration

Our work handles the variabilities at the infrastructure layer, and similar to [74], we use feature modeling to select the resources according to users' requirements. However, our method differs from these works in the following ways: (i) it addresses the configuration options at the IaaS layer independent of cloud provider, and in a way that different user profiles could express their preferences, enabling model reuse; (ii) it considers multiple service selections (e.g., VM,

VMI, storage); (iii) it uses multiple feature models to handle the configuration options of inter-cloud environments, without demanding pre-configured virtual machine image.

7.2 Autonomic cloud systems

Over the years, different cloud systems have been proposed in the literature, with different objectives. In Table 9, we present a comparative view of important features of these systems. The system is presented in the first column. The second column presents if the available system implements self-configuration, which means that it can select and configure the resources automatically. The third column presents if it implements failure recovery policy. The fourth and the fifth columns show if the systems are context-aware and if they implement self-optimization. Context-aware in this case, means if nodes are organized considering their location. The cloud model of the system is presented in the sixth column. Finally, the last column presents if the system can work in an inter-cloud environment.

As can be seen, half of the works implement self-configuration properties. In this case, the developers specify their needs in a service manifest, and the system automatically selects a cloud provider and deploys the applications. Moreover, self-healing property is implemented by some of the works. For example, Snooze [20] employs a hierarchical resource management, and uses multicast to locate the different nodes (e.g., group managers, local controllers). Similar to us, CometCloud [37] organizes the nodes using a P2P overlay to connect and to detect failures. However, it organizes the nodes unaware of their locations.

Only two systems [21,37] deploy the applications taking into account the location of the resources (i.e., nodes) where the application will run. For example, OPTMIS [21] deploys the applications taking into account the energy consumption of the cloud as well its carbon footprint. Another example is CometCloud [37]. CometCloud distributes the tasks considering three security domain level. In this case, there is a master that tries to send the tasks to trusted workers, i.e., workers that are running in a private cloud. Tasks are sent to untrusted workers (i.e., workers that are running in a public cloud) only when the SLA violation ratio exceeds a threshold.

Moreover, the majority of the systems, implements self-optimization property to help the developers on meeting the (QoS) constraints of their native cloud applications [41, 51,56,57] or to minimize power consumption [20].

Finally, only two systems target IaaS cloud, and the other ones (PaaS) cloud.

CometCloud [37] is the closest work to ours. Our work differs from it in the following ways. First, we consider that self-configuration is an important feature to support the users on running their applications in the clouds, since most of the users do not have cloud and/or system administration skills.

Our self-configuration approach relies on feature models (FMs), to deal with heterogeneous cloud providers' resource descriptions, and also to enable a declarative strategy. This frees the users from the work of creating pre-configured virtual machine images on each cloud. Second, we use a hierarchical P2P overlay to organize the resources, which helps us to reduce the cost of communication between the clouds and to keep the distributed hash table (DHT) up-to-date. In CometCloud, the resources are organized in the same P2P overlay, and it uses different security domains to distribute the tasks. However, this work does no implement self-optimization feature since it requires advanced knowledge about the workload/application.

Table 9 Comparison of cloud architectures considering their autonomic properties

Architecture	SC	SH	CA	SO	Cloud model	Inter-clouds
Cloud-TM [57]	No	Yes	No	Yes	PaaS	Yes
JSTaaS [41]	Yes	No	No	Yes	PaaS	Yes
mOSAIC [51]	Yes	No	No	Yes	PaaS	Yes
Reservoir [56]	Yes	Yes	No	Yes	PaaS	Yes
OPTIMIS [21]	Yes	No	Yes	Yes	PaaS	Yes
FraSCaTi [61]	Yes	No	No	No	PaaS	Yes
TClouds [73]	No	Yes	No	No	PaaS	Yes
COS [30]	No	No	No	Yes	PaaS	No
Snooze [20]	No	Yes	No	Yes	IaaS	No
CometCloud [37]	No	Yes	Yes	Yes	IaaS	Yes
Dohko	**Yes**	**Yes**	**Yes**	**No**	**IaaS**	**Yes**

SC self-configuration, *SH* self-healing, *CA* context-awareness, *SO* self-optimization

8 Conclusion

Using multiple IaaS clouds is still a challenging and time-consuming activity, even for experienced system administrators. The difficulties mostly exist since IaaS clouds offer low-level access to the infrastructure resources. Thus, the users are responsible for selecting, configuring and managing the computing resources on the clouds. Likewise, the clouds target web applications, whereas the users' applications are often batch-oriented systems. Hence, this can be a barrier to use clouds' services.

Therefore, in this paper, we presented and evaluated *Dohko*, an autonomic and goal-oriented system for provisioning and management of inter-cloud environments. *Dohko* enables a declarative strategy. In this case, the users submit to the system a corresponding specification of the comput-

ing environment configuration they want. Then, the system automatically creates and configures the whole computing environments, taking into account users' requirements. Finally, it executes the applications in one or across multiple clouds.

Our system implements the autonomic properties [29]: self-configuration, self-healing, and context-awareness. Likewise, it relies on a hierarchical P2P overlay to organize the nodes distributed across various clouds. Finally, the self-configuration property is based on a software product line engineering (SPLE) method to handle the variability options of the clouds. In this case, our SPLE method defines an *abstract feature model* to handle the commonalities of the clouds, many *concrete feature models* to describe the configuration options of each cloud, and a *configuration knowledge (CK)* to map how the products are instantiated from these models.

The advantages of our SPLE method are manifold. First, it avoids invalid configurations or configurations that do not match some objectives, without requiring from users cloud or system administration skills. Second, it provides a uniform view of the clouds, translating specific cloud terms to concepts independent of cloud providers. Third, it can support the users on provisioning their resources based on multiple parameters such as the location where the resources should be deployed; the software packages, or the cloud provider. Fourth, it can be used to document inter-cloud environments without demanding pre-configured virtual machine images.

Experimental results on two different cloud providers show that with our solution, ordinary users may have access to inter-cloud computing environments, that when configured by hands demands detailed cloud computing skills.

In our experiments, we could execute the SSEARCH application to compare up to 24 query sequences with the database UniProtKB/Swiss-Prot in two distinct cloud providers—Amazon EC2 and Google Compute Engine (GCE)—considering five execution scenarios. This approach helps on tackling the lack of middleware prototypes that can support different scenarios when executing applications across multiple clouds. Furthermore, since *Dohko* is essentially a self-management system, it frees the users from the details of computing system operations, allowing them to concentrate on their objectives rather than spending their time managing the computing environments.

One issue of our system is the lack of a self-optimization strategy since it does not focus on task scheduling. Furthermore, our approach does not consider pre-packaged configuration descriptions, such as Docker Hub Images (*hub.docker.com*) or Ansible Galaxy (*galaxy.ansible.com*), and we leave these extensions as our future work. The software prototype is available at *dohko.io*.

Acknowledgements The authors would like to thank CAPES/CNPq/Brazil, Inria Saclay/France (project POSTALE), and MINES ParisTech/France for their financial support.

References

1. Adams, K., Agesen, O.: A comparison of software and hardware techniques for x86 virtualization. ACM SIGOPS Oper. Syst. Rev. **40**(5), 2–13 (2006)
2. Almeida, A., Cavalcante, E., Batista, T., Cacho, N., Lopes, F., Delicato, F.C., Pires, P.F.: Dynamic adaptation of cloud computing applications. In: 25th SEKE, pp. 67–72 (2013)
3. Androutsellis-Theotokis, S., Spinellis, D.: A survey of peer-to-peer content distribution technologies. ACM Comput. Surv. **36**(4), 335–371 (2004)
4. Arabshian, K., Schulzrinne, H.: Distributed context-aware agent architecture for global service discovery. In: 2nd SWUMA (2006)
5. Basili, V.R., Caldiera, G., Rombach, H.D.:. The goal question metric approach. In: Encyclopedia of Software Engineering. Wiley, New York (1994)
6. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated analysis of feature models 20 years later: a literature review. Inf. Syst. **35**(6), 615–636 (2010)
7. Casanova, H., Zagorodnov, D., Berman, F., Legrand, A.:. Heuristics for scheduling parameter sweep applications in grid environments. In: 9th HCW, pp. 349–363 (2000)
8. Casavant, T.L., Kuhl, J.G.: A taxonomy of scheduling in general-purpose distributed computing systems. IEEE Trans. Softw. Eng. **14**(2), 141–154 (1988)
9. Cavalcante, E., Almeida, A., Batista, T., Cacho, N., Lopes, F., Delicato, F.C., Sena, T., Pires, P.F.: Exploiting software product lines to develop cloud computing applications. In: 16th SPLC, pp. 179–187 (2012)
10. Celesti, A., Tusa, F., Villari, M.: Toward cloud federation: concepts and challenges. In: Brandic, I., Villari, M., Tusa, F. (eds.) Achieving Federated and Self-manageable Cloud Infrastructures: Theory and Practice, pp. 1–17. IGI Global, Hershey (2012)
11. Chieu, T.C., Mohindra, A., Karve, A.A., Segal, A.: Solution-based deployment of complex application services on a cloud. In: IEEE SOLI, pp. 282–287 (2010)
12. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Boston (2001)
13. Coulouris, G., Dollimore, J., Kindberg, T., Blair, G.: Distributed Systems: Concepts and Design, 5th edn. Addison-Wesley, Boston (2011)
14. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. ACM Press/Addison-Wesley, Boston (2000)
15. Czarnecki, K., Helsen, S., Eisenecker, U.: Formalizing cardinality-based feature models and their specialization. Softw. Process **10**(1), 7–29 (2005a)
16. Czarnecki, K., Helsen, S., Ulrich, E.: Staged configuration through specialization and multilevel configuration of feature models. Softw. Process **10**, 143–169 (2005b)
17. Davis, D., Pilz, G.: Cloud infrastructure management interface (CIMI) model and RESTful HTTP-based protocol: An interface for managing cloud infrastructure. Technical Report DSP0263, Distributed Management Task Force, May 2012. http://bit.ly/2msdlF7. Accessed Jan 2017
18. Dougherty, B., White, J., Schmidt, D.C.: Model-driven auto-scaling of green cloud computing infrastructure. FGCS **28**(2), 371–378 (2012)

19. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Comput. Surv. **35**(2), 114–131 (2003)
20. Feller, E., Rilling, L., Morin, C.: Snooze: a scalable and autonomic virtual machine management framework for private clouds. In: IEEE/ACM CCGrid, pp. 482–489 (2012)
21. Ferrer, A.J., Hernández, F., Tordsson, J., Elmroth, E., Ali-Eldin, A., Zsigri, C., Sirvent, R., Guitart, J., Badia, R.M., Djemame, K., Ziegler, W.: OPTIMIS: a holistic approach to cloud service provisioning. FGCS **28**(1), 66–77 (2012)
22. Forum, Global Inter-Cloud Technology: Use cases and functional requirements for inter-cloud computing. Technical report, Global Inter-Cloud Technology Forum August (2010)
23. García-Galán, J., Rana, O., Trinidad, P., Ruiz-Cortés, A.: Migrating to the cloud: a software product line based analysis. In: 3rd CLOSER, pp, 416–426 (2013)
24. Graham, R.L., Lawler, E.L., Lenstra, J.K., Kan, A.R.: Optimization and approximation in deterministic sequencing and scheduling: a survey. Ann. Discret. Math. **5**, 287–326 (1979)
25. Grozev, N., Buyya, R.: Inter-Cloud architectures and application brokering: taxonomy and survey. Software **44**(3), 369–390 (2014)
26. Hariri, N., Castro-Herrera, C., Mirakhorli, M., Cleland-Huang, J., Mobasher, B.: Supporting domain analysis through mining and recommending features from online product listings. IEEE Trans. Softw. Eng. **39**(12), 1736–1752 (2013)
27. Hayashibara, N., Cherif, A.:. Failure detectors for large-scale distributed systems. In: 21st IEEE SRDS, pp. 404–409 (2002)
28. Hoefer, C.N., Karagiannis, G.: Taxonomy of cloud computing services. In: IEEE GLOBECOM Workshops, pp. 1345–1350 (2010)
29. Horn, P.: Autonomic computing: IBM's perspective on the state of information technology. http://bit.ly/2excwlX2001. Accessed Jan 2017
30. Imai, S., Chestna, T., Varela, C.A.: Elastic scalable cloud computing using application-level migration. In: 5th IEEE/ACM UCC, pp. 91–98 (2012)
31. Iosup, A., Ostermann, S., Yigitbasi, M.N., Prodan, R., Fahringer, T., Epema, D.: Performance analysis of cloud computing services for many-tasks scientific computing. IEEE TPDS **22**(6), 931–945 (2011)
32. Jackson, K.R., Ramakrishnan, L., Muriki, K., Canon, S., Cholia, S., Shalf, J., Wasserman, H.J., Wright, N.J.: Performance analysis of high performance computing applications on the Amazon Web Services cloud. In: 2nd IEEE CloudCom, pp. 159–168 (2010)
33. Jussien, N., Rochart, G., Lorca, X.: Choco: an Open Source Java Constraint Programming Library. In: OSSICP, pp. 1–10 (2008)
34. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, November 1990. http://bit.ly/2mJX4aD. Accessed Jan 2017
35. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Comput. **36**(1), 41–50 (2003)
36. Kermarrec, A.M., Triantafillou, P.: XL peer-to-peer pub/sub systems. ACM Comput. Surv. **46**(2), 16 (2013)
37. Kim, H., Parashar, M.: CometCloud: An Autonomic Cloud Engine. Wiley, New York (2011)
38. Konstantinou, A.V., Eilam, T., Kalantar, M., Totok, A.A., Arnold, W., Snible, E.: An architecture for virtual solution composition and deployment in infrastructure clouds. In: 3rd VTDC, pp. 9–18 (2009)
39. Leite, A.F., de Melo, A.C.: Executing a biological sequence comparison application on a federated cloud environment. In: 19th HiPC, pp. 1–9 (2012)
40. Leite, A.F., Alves, V., Rodrigues, G.N., Tadonki, C., Eisenbeis, C., de Melo, A.C.: Automating resource selection and configuration in inter-clouds through a software product line method. In: 8th IEEE CLOUD, pp. 726–733 (2015)
41. Leitner, P., Rostyslav, Z., Gambi, A., Dustdar, S.: A framework and middleware for application-level cloud bursting on top of infrastructure-as-a-service clouds. In: 6th IEEE/ACM UCC, pp. 163–170 (2013)
42. Lewis, G.A.: Role of standards in cloud-computing interoperability. In: 46th HICSS, pp. 1652–1661 (2013)
43. Lewis, J., Fowler, M.: Microservices: a definition of this new architectural term. http://bit.ly/1dI7ZJQ 2014
44. Li, Q., Li, H., Russell, P., Cheng, Z., Wang, C.: CA-P2P: context-aware proximity-based peer-to-peer wireless communications. IEEE Commun. Magazine **52**(6), 32–41 (2014)
45. Di Martino, B.: Applications portability and services interoperability among multiple clouds. IEEE Cloud Comput. **1**(1), 74–77 (2014)
46. Mietzner, R., Metzger, A., Leymann, F., Pohl, K.: Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications. In: PESOS, pp. 18–25 (2009)
47. OASIS Standard. Topology and orchestration specification for cloud applications version 1.0. http://bit.ly/1A6D4C8 May 2013
48. Ostermann, S., Iosup, A., Yigitbasi, N., Prodan, R., Fahringer, T., Epema, D.: A performance analysis of EC2 cloud computing services for scientific computing. In: Cloud Computing, pp. 115–131. Springer, New York (2010)
49. Ou, Z., Zhuang, H., Lukyanenko, A., Nurminen, J.K., Hui, P., Mazalov, V., Yla-Jaaski, A.: Is the same instance type created equal? exploiting heterogeneity of public clouds. IEEE Trans. Cloud Comput. **1**(2), 201–214 (2013)
50. Petcu, D.: Consuming resources and services from multiple clouds. JGC **12**, 1–25 (2014)
51. Petcu, D., Craciun, C., Rak, M.: Towards a cross platform cloud API. In: 1st CLOSER, pp. 166–169 (2011)
52. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, New York (2005)
53. Prodan, R., Ostermann, S.:. A survey and taxonomy of infrastructure as a service and web hosting cloud providers. In: 10th IEEE/ACM CCGrid, pp. 17–25 (2009)
54. Quinton, C., Haderer, N., Rouvoy, R., Duchien, L.: Towards multi-cloud configurations using feature models and ontologies. In: MultiCloud, pp. 21–26 (2013)
55. Quinton, C., Romero, D., Duchien, L.: Automated selection and configuration of cloud environments using software product lines principles. In: 7th IEEE CLOUD (2014)
56. Rochwerger, B., Breitgand, D., Levy, E., Galis, A., Nagin, K., Llorente, I.M., Montero, R., Wolfsthal, Y., Elmroth, E., Caceres, J., Ben-Yehuda, M.: The Reservoir model and architecture for open federated cloud computing. IBM J. Res. Dev. **53**(4), 1–4 (2009)
57. Romano, P., Rodrigues, L., Carvalho, N., Cachopo, J.: Cloud-TM: Harnessing the cloud with distributed transactional memories. ACM SIGOPS Oper. Syst. Rev. **44**(2), 1–6 (2010)
58. Ruehl, S.T., Andelfinger, U.: Applying software product lines to create customizable software-as-a-service applications. In: 15th SPLC, p. 16 (2011)
59. Ruehl, S.T., Andelfinger, U., Rausch, A., Verclas, S.A.: Toward realization of deployment variability for software-as-a-service applications. In: 5th IEEE CLOUD, pp. 622–629 (2012)
60. Schroeter, J., Mucha, P., Muth, M., Jugel, K., Lochau, M.: Dynamic configuration management of cloud-based applications. In: 16th SPLC, pp. 171–178 (2012)
61. Seinturier, L., Merle, P., Fournier, D., Dolet, N., Schiavoni, V., Stefani, J.B.: Reconfigurable SCA Applications with the FraSCAti Platform. In: IEEE SCC, pp. 268–275 (2009)
62. Sill, A.: The role of communities in developing cloud standards. IEEE Cloud Comput. **1**(2), 16–19 (2014)

63. Silvestro, J., Canavese, D., Cesena, E., Smiraglia, P.: A unified ontology for the virtualization domain. In: OnTheMove, pp. 617–624 (2011)
64. Smith, T.F., Waterman, M.S.: Identification of common molecular subsequences. J. Mol. Biol. **147**, 195–197 (1981)
65. Sousa, G., Rudametkin, W., Duchien, L.: Automated setup of multi-cloud environments for microservices applications. In: 9th IEEE CLOUD, pp. 327–334 (2016)
66. Stantchev, V.: Performance evaluation of cloud computing offerings. In: 3rd ADVCOMP, pp. 187–192 (2009)
67. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup service for internet applications. In: SIGCOMM, pp. 149–160 (2001)
68. Sun, X., Tian, Y., Liu, Y., He, Y.: An unstructured P2P network model for efficient resource discovery. In: ICADIWT, pp. 156–161 (2008)
69. Le Nhan T., Sunyé, G., Jezequel, J.M.: A model-based approach for optimizing power consumption of IaaS. In: 2nd NCCA, pp. 31–39 (2012a)
70. Le Nhan T., Sunyé, G., Jézéquel, J.M.: A model-driven approach for virtual machine image provisioning in cloud computing. In:1st ESOCC, pp. 107–121 (2012b)
71. Tang, T., Yew, P.C.: Processor self-scheduling for multiple-nested parallel loops. In: ICPP, pp. 528–535 (1986)
72. Toosi, A.N., Calheiros, R.N., Buyya, R.: Interconnected cloud computing environments: challenges, taxonomy, and survey. ACM Comput. Surv. **47**(1), 7 (2014)
73. Verissimo, P., Bessani, A., Pasin, M.: The TClouds architecture: open and resilient cloud-of-clouds computing. In: IEEE/IFIP 42nd DSN, pp. 1–6 (2012)
74. Walraven, S., Van Landuyt, D., Truyen, E., Handekyn, K., Joosen, W.: Efficient customization of multi-tenant software-as-a-service applications with service lines. JSS **91**, 48–62 (2014)
75. Wittern, E., Kuhlenkamp, J., Menzel, M.: Cloud service selection based on variability modeling. In: 10th ICSOC, pp. 127–141 (2012)
76. Yau, S.S., Wang, Y., Karim, F.: Development of situation-aware application software for ubiquitous computing environments. In: 26th COMPSAC, pp. 233–238 (2002)
77. Youseff, L., Butrico, M., da Silva, D.: Toward a unified ontology of cloud computing. In: GCE, pp. 1–10 (2008)
78. Zhang, T., Du, Z., Chen, Y., Ji, X., Wang, X.: Typical virtual appliances: an optimized mechanism for virtual appliances provisioning and management. JSS **84**(3), 377–387 (2011)
79. Zoels, S., Despotovic, Z., Kellerer, W.: On hierarchical DHT systems—an analytical approach for optimal designs. Comput. Commun. **31**(3), 576–590 (2008)

Alessandro Ferreira Leite received his PhD in Computer Science from the Paris-Sud University, France and from the University of Brasilia (UnB), Brazil, in 2014, his MSc in Computer Science from University of Brasilia (UnB), Brazil, in 2010 and his BSc in Computer Science from University Center of Brasilia, Brazil, in 2008. Among his research interests include distributed systems, high-performance computing, data mining, and machine learning. He is member of the Association for Computer Machinery (ACM), the IEEE Society, and IEEE Computer Society.

Vander Alves is Adjunct Professor at the Computer Science Department of University of Brasilia, Brazil. He conducts research on Software Product Lines, Ambient Assisted Living, and Body Sensor Networks. Currently, he is also a CAPES-Humboldt Fellow at the University of Passau, Germany. Previously he held research and development positions at Fraunhofer IESE, Lancaster University, and IBM Silicon Valley Lab.

Genaína Nunes Rodrigues is a tenure professor in the Department of Computer Science at the University of Brasilia. She received her PhD in Computer Science from University College London (2008) in the Software Engineering group. Previously, she obtained her bachelor's degree in Computer Science from the University of Brasilia in 1999 and her Master's Degree in Computer Science from the Federal University of Pernambuco (2002) in the Distributed Systems group. Her research interests are mostly in Software Systems Engineering including dependability analysis and modeling, self-adaptive systems, goal-oriented requirements engineering, verification of probabilistic models, model-driven development for quality requirements, as well as distributed platforms. In 2013, she co-organized the Brazilian Software Engineering Congress (CBSoft) of the Brazilian Computer Society (SBC). Together with her research group, LADECIC, she has engaged national and international cooperations in various aspect of her topics of research interest.

Claude Tadonki is a senior researcher and lecturer at the MINES ParisTech Institute (Paris/France) since 2011. He holds a PhD and a Habilitation degree in computer science from University of Rennes and from Paris-Sud University respectively. After six years of cutting-edge research in operational research and theoretical computer science at the University of Geneva, he relocated to France to work for EMBL, University of Paris-Sud, LAL-CNRS and then MINES ParisTech. His main research topics included High Performance Computing, and Operational Research. He has worked at several laboratories and universities, has initiated various scientific

projects and national/international collaborations, has given significant number of CS courses in different contexts including industries. He acts as a reviewer for high-impact international journals and major conferences, and he has published numerous papers in journals and conferences. He has participated and co-organized number of HPC conferences and meetings around the world.

Christine Eisenbeis is an Inria researcher at the Paris-Sud University, Orsay. She received her PhD degree in 1986 on loop optimization for array-processors at the Paris 6 University in 1986. Her research interests include compiler technologies for high performance microprocessors, instruction scheduling, software pipelining, register allocation, data dependence analysis and data locality optimization. She has also worked on compiler/architecture codesign for video coprocessors with special emphasis on time specification in programs. She is now working on new computation and programming models in relation with complex systems and information theory, including reversible computing, as well as on the impact of digital technology on work.

Alba Cristina Magalhaes Alves de Melo received her PhD in Computer Science from the Institut National Polytechnique de Grenoble (INPG), France, in 1996, her MSc in Computer Science from Federal University of Rio Grande do Sul, Brazil, in 1991 and her BSc in Computer Science from University of Brasilia, Brazil, in 1986. She is currently Full Professor in the Computer Science Department at the University of Brasilia, Brazil, and CNPq Research Fellow level 1D. Her research interests include cloud computing, high performance computing and computational biology. She is a senior member of the IEEE Society and IEEE Computer Society.