Model-Driven Elasticity Management with OCCI

Yahya Al-Dhuraibi, Faiez Zalila, Nabil Djarallah, Philippe Merle

Model-Driven Elasticity Management with OCCI

Yahya Al-Dhuraibi, Faiez Zalila, Nabil Djarallah, Philippe Merle

Abstract—Elasticity is considered as a fundamental feature of cloud computing where the system capacity can adjust to the current application workloads by provisioning or de-provisioning computing resources automatically and timely. Many studies have been already conducted to elasticity management systems, however, almost all lack to offer a complete modular solution. In this article, we propose MODEMO, a new elasticity management system powering both vertical and horizontal elasticities, both VM and Container virtualization technologies, multiple cloud providers simultaneously, and various elasticity policies. MODEMO is characterized by the following features: it represents (*i*) the first system that manages elasticity using Open Cloud Computing Interface (OCCI) model with respect to the OCCI standard specifications, (*ii*) the first unified system which combines the functionalities of the worldwide cloud providers: Amazon Web Services (AWS), Microsoft Azure and Google Cloud Platform (GCP), and (*iii*) allows a dynamic configuration at runtime during the execution of the application. MODEMO permits to timely adapt resource capacity according to the workload intensity and increase application performance without introducing a significant overhead.

Index Terms—Cloud computing; Elasticity; Open Cloud Computing Interface; Containers; Virtual machines; Model-driven engineering.

1 Introduction

CLOUD computing has become a preferable solution for deploying applications and services. These applications require a variable amount of computing resources depending on the changing workload intensity at runtime. In addition, due to the heated marketplace competition in the cloud domain, providers have been under pressure to produce attractive services that satisfy customers by maintaining applications performance and respecting the Service Level Agreement (SLA) with optimal costs. Therefore, elasticity is a vital asset as it allows to increase or decrease the capacity of virtual resources timely according to the need [1]. There are two main approaches of elasticity: vertical and horizontal. Vertical elasticity consists of increasing or decreasing characteristics of computing resources, such as CPU cores, memory, network bandwidth, etc. Horizontal elasticity is the process of adding/removing resource instances, which may be located at different locations. Load balancers are used to distribute the load among the different instances. Since elasticity is a key feature in cloud computing, it has been widely explored by many works. For example, the works [2], [3] [4], and [5] address the vertical elasticity, while the works [6] and [7] focus on the horizontal elasticity. Cloud elasticity is diverse and heterogeneous because it has different approaches, policies, purposes, and applications [1]. There are different policies that the elasticity controller can use to decide when and how to provision or deprovision the resources. Elasticity has also different purposes such as improving performance, increasing resource capacity, saving energy, reducing cost and ensuring availability. In addition, elasticity can be applied at the infrastructure level or application level. The infrastructure is powered by a certain virtualization technology such as VMware, Xen, or a provider-specific virtualization platform. Container, a lightweight virtualization technology, is also increasingly adopted by the cloud providers. To the best of our knowledge, there is no work that proposes an elasticity management system (EMS) supporting both vertical and horizontal elasticities, both VM and container, multiple cloud providers, and various elasticity policies. The contribution of this article is to present a new elasticity management system called MODEMO (Model-Driven Elasticity Management with OCCI). The main characteristics of MODEMO are:

- **Standard-based**: Most of the existing EMSs are proprietary systems, MODEMO is based on Open Cloud Computing Interface (OCCI) standard.
- **Model-Driven**: Most of the existing EMSs are defined by an API while we propose a high-level model that describes all aspects of elasticity.
- **Both vertical and horizontal elasticities**: The majority of EMSs support only one type of elasticity. MODEMO supports both.
- **Both VM and container**: MODEMO supports both VM and container virtualization technologies.
- **Multi-cloud providers**: Most existing EMSs are dedicated to a particular cloud provider. MODEMO supports different cloud providers simultaneously.
- **Multiple elasticity policies**: MODEMO supports different categories of elasticity policies such as scaling, scheduling, migration, and swapping policies. MODEMO is the first EMS that combines all the elasticity policies of the most popular cloud providers including Microsoft Azure, Amazon Web Services (AWS), and Google Cloud Platform (GCP). Based on the chosen policy, MODEMO permits to adapt resource capacity and increase application performance.
- **Highly extensible**: New elasticity policies, provider allo-

- *Y. Al-Dhuraibi is with Scalair company, Hem, France. E-mail: yaldhuraibi@scalair.fr*
- *F. Zalila is with Inria Lille - Nord Europe, Villeneuve d'Ascq, France. E-mail: faiez.zalila@inria.fr*
- *N. Djarallah is with Scalair company, Hem, France. E-mail: ndjarallah@scalair.fr*
- *P. Merle is with Inria, Lille - Nord Europe, Villeneuve d'Ascq, France. E-mail: philippe.merle@inria.fr*

Manuscript received June 15, 2018.

cation policies, load balancer algorithms, monitored metrics, etc. can be added easily. MODEMO eases the utilization of the elasticity controller. The elasticity mechanisms and metrics are built with plug in/out facilities.
- **Highly reconfigurable**: MODEMO is an EMS reconfigurable during the execution. MODEMO allows elasticity controller settings at runtime through Models@run.time approach [8]. Consequently, the elasticity controller applies these configurations immediately.
- **Negligible overhead**: MODEMO introduces a negligible overhead.

The remainder of this article is organized as follows. Section 2 describes the motivation for this work. Section 3 presents the background on OCCI. Section 4 presents our MODEMO system and its different components. Section 5 presents the validation of our solution. Section 6 provides a short discussion about this work. After that, we discuss related works in Section 7. Section 8 concludes the article and highlights research perspectives.

2 MOTIVATION

In this section, we investigate a motivating real cloud scenario and specific limitations among existing Elasticity Management Systems (EMSs).

Motivating Scenario. Scalair[1] provides cloud resources using its own technologies such as VMware vSphere and OpenStack for private clouds. At the same time, Scalair acts as a partner with other public providers such as AWS and Microsoft Azure, which makes it a multi-cloud provider. Its infrastructure is mainly enabled with different virtualization computing units, VMs and containers. The hosted customers' applications require different types of elasticity. The elasticity is derived using different policies depending on the customer objective and the cloud provider. This framework should respect a certain standard in order to facilitate its usage by the different stakeholders in the company, and also it should cover all the aspects of elasticity. Scalair needs a framework to manage such diversity and heterogeneity. Therefore, we proposed a unified system (MODEMO) to address these needs. MODEMO is characterized by many features that make it different from the other existing works.

- **Standard-based**: Most of the existing EMSs that manage cloud elasticity are proprietary systems which introduce vendor lock-in problem, they are not based on a standard while MODEMO is based on OCCI standard. OCCI is an extremely important paradigm that defines open standard API specifications in the cloud space [9].
- **Model-Driven**: Despite the numerous works involved in elasticity and resource management, there is almost no model-based framework that can manage all aspects of elasticity. Most existing EMSs are defined by an API while we propose, in this article, a high-level model that describes all aspects of elasticity. Our approach employs the Model-Driven Engineering (MDE) approach [10] in order to handle elasticity resources and controller policies at a higher level of abstraction based on the OCCI standard.
- **Vertical and horizontal elasticities**: The majority of elasticity solutions support only one type of elasticity: horizontal xor vertical. Only a few EMSs support both of them [6], [7]. MODEMO supports both types of elasticity.
- **VMs and containers**: Most works concentrates on VM elasticity and few works address the elasticity of containers [1]. MODEMO supports both virtualization technologies: VMs and containers.
- **Multi-cloud providers**: Most existing EMSs are dedicated to a particular provider. Few works [11], [12] support multiple clouds, however, the chosen provider must be defined manually. MODEMO supports different providers simultaneously. In addition, MODEMO has an allocation policy that permits to seamlessly and automatically lease the resource from different providers.
- **Multiple elasticity policies**: Since the elasticity is heterogeneous, it has diverse mechanisms and objectives, each work around elasticity concentrates on a single approach for a certain purpose. MODEMO is the first modular approach that gives the possibility to use multiple elasticity policies in a single system. MODEMO provides the elasticity policies supported by the most popular cloud providers and more. These providers include Microsoft Azure, AWS, and GCP. MODEMO not only supports many elasticity policies including policies supported by these providers but also provides an improvement for the existing ones. For example, simple dynamic scaling policy in AWS only support one metric while MODEMO permits combining many metrics in the same policy according to a certain logic. We will explain the different policies and the added value or improvement in the elasticity controller policies in Section 4. Generally, the variability and heterogeneity of the elasticity horizon (techniques, approaches, purposes, etc.) are very large. MODEMO is a unified modular solution for these issues.
- **Highly extensible**: MODEMO provides a separation in the abstraction between resources and policies. Various policies can be added dynamically. Most existing EMSs are introduced as a single entity or blackbox. The Elasticity controller in MODEMO is loosely coupled where many components support drag-and-drop (plugins) functionality.
- **Highly reconfigurable**: There are many parameters and settings which determine the behavior of an elasticity controller, *e.g.,* the maximum number of instances in a horizontal group, the memory maximum size or vCPUs of a compute instance. Examples of other parameters are: *cool duration*, *upper threshold*, *lower threshold*, etc. MODEMO allows changing such settings at runtime. Different policies such as *AllocationPolicy*, *LoadBalancerPolicy*, etc. can be changed dynamically. As discussed in Section 4, a rule is used to evaluate a metric (CPU, memory, etc) which can be modified at runtime and, thus, leads to change the monitoring system and the elasticity policy behavior at runtime.

3 BACKGROUND

A brief description of the OCCI standard and the OCCIware toolchain are provided in this section since our MODEMO model is based on OCCI standard and implemented using the OCCIware toolchain. As MODEMO is a framework for

1. http://www.scalair.fr

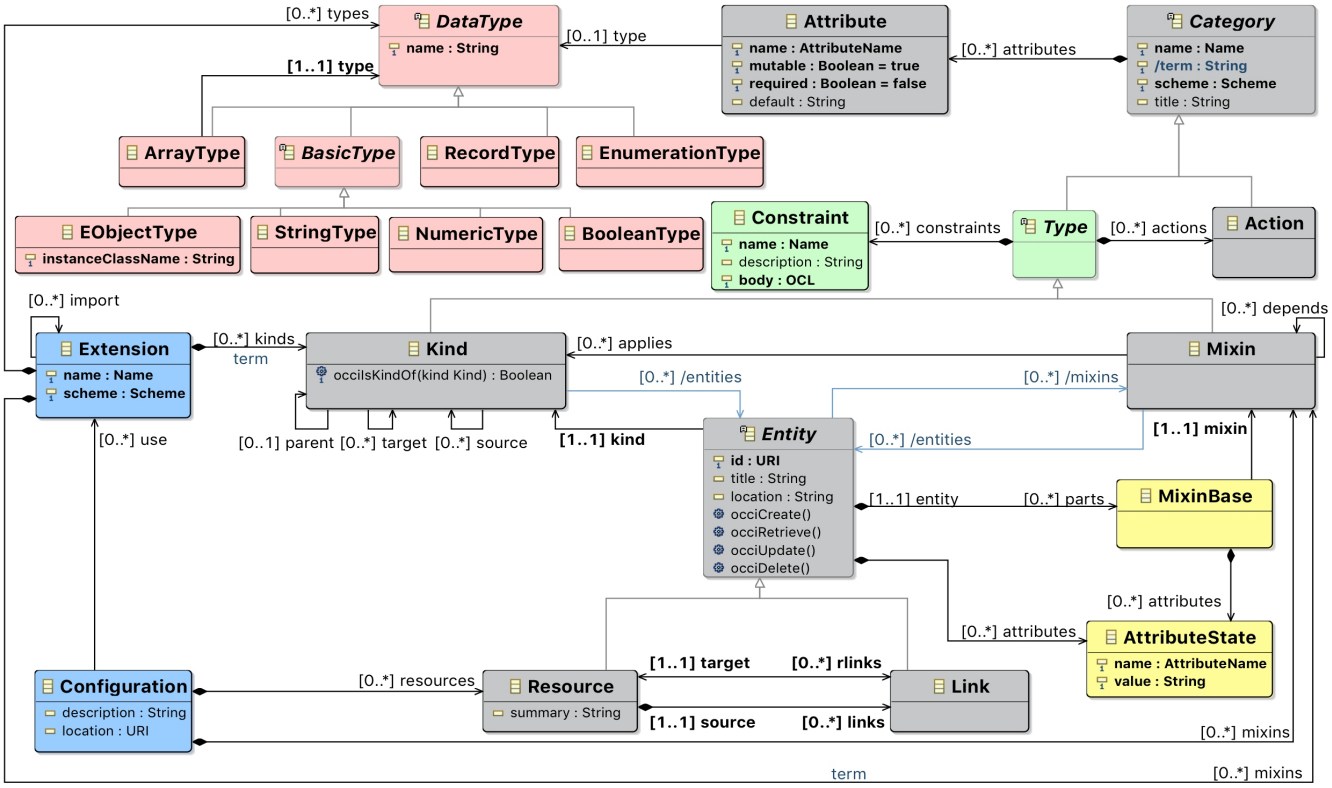


Fig. 1: The OCCIware metamodel

managing elasticity aspects in the context of cloud environments, we have chosen to use OCCI. OCCI is the only open cloud standard providing a general-purpose model for cloud resources and a RESTful API for efficiently accessing and managing any kind of these resources, e.g., Infrastructure as a Service, Platform as a Service, and Software as a Service. This will solve the interoperability challenge between clouds, as providers will be specified by the same resource-oriented model called the OCCI Core Model [13]. OCCI model allows to abstract cloud concepts and to manage these modeled resources at runtime. OCCI was introduced by Open Grid Forum (OGF) for managing any kind of cloud resources [9]. OCCI is delivered as a set of specification documents divided into the four following categories: OCCI Core Model, OCCI Protocols, OCCI Renderings, and OCCI extensions.

The OCCI Core model [14] is composed of eight elements (grey boxes in Figure 1): *Resource*, *Link*, *Entity*, *Kind*, *Mixin*, *Action*, *Category*, and *Attribute*. *Resource* is the root abstraction of any cloud resource such as a virtual machine, a database, etc. *Link* represents a relation between two resources, such as a virtual machine connected to a network and an application hosted by a virtual machine. *Entity* is an abstract base class inherited by both *Resource* and *Link*. *Kind* is the concept of type within OCCI such as Compute, Network, and Container. *Mixin* represents a set of attributes and actions that can be dynamically plugged in/out to an OCCI entity. *Mixin* can be applied to zero or more kinds and can depend on zero or more other *Mixin* instances. *Action* represents a business-specific behavior that can be

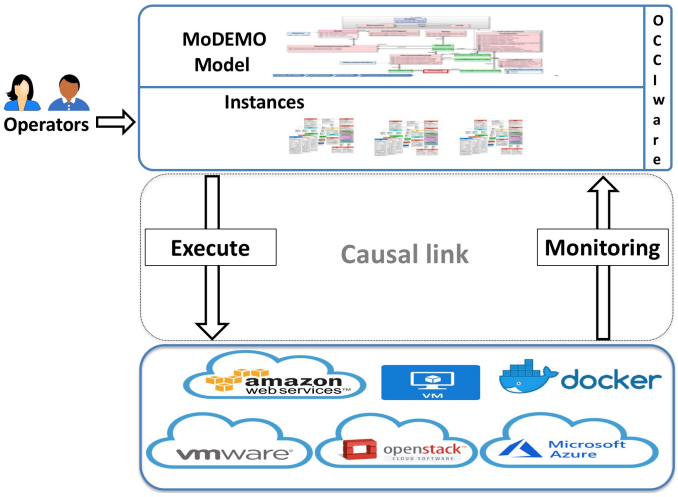


Fig. 2: MoDEMO overview

executed on entities such as start/stop a VM. *Category* is an abstract base class inherited by *Kind*, *Mixin*, and *Action*. *Attribute* defines a client-visible property, *e.g.*, the IP address of a network.

During the OCCIware project, a precise metamodel of OCCI has been proposed, named OCCIware metamodel [15]. OCCIware is a research and development project funded by French Programme d'Investissements d'Avenir (PIA). This project aims at building a comprehensive, yet modular software engineering toolchain dedicated

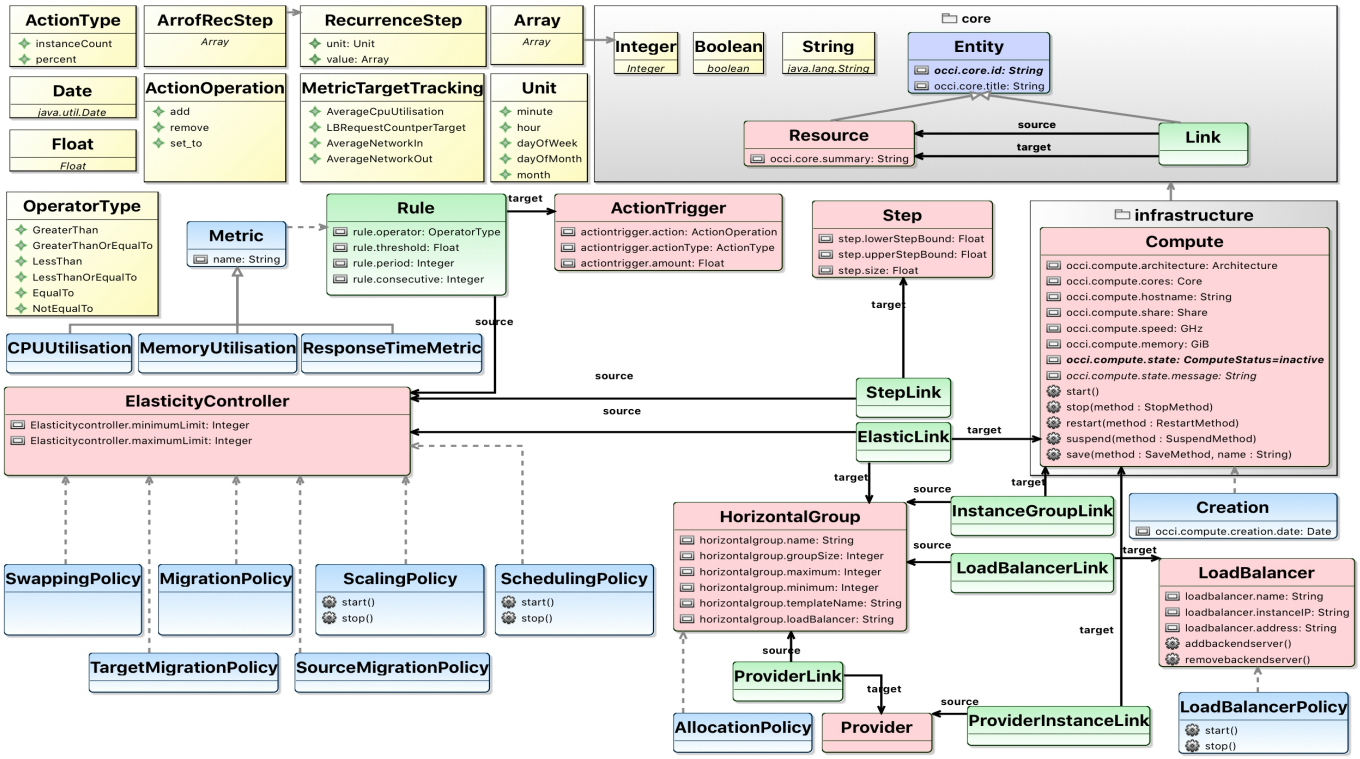


Fig. 3: MoDEMO Model

to service-oriented applications to offer a unified interface for the different clouds [16]. All the concepts of the OCCIware Metamodel are introduced and modeled in [13]. In the following, we detail a subset of the OCCIware metamodel required to understand the rest of this article. *Extension* represents a cloud domain such as infrastructure, platform, etc. An *Extension* has a *name* and a *scheme*. It owns a set of *kinds*, *mixins* and *types*, and can import zero or more *extensions*. *Configuration* represents a running OCCI system. It owns zero or more *resources* (and transitively *links*), and use zero or more *extensions*. *DataType* is the root of a data type system for OCCI as shown at the left part of Fig. 1 (the red-colored classes). This data type system allows us to define primitive types such as *StringType* to model string types, *NumericType* to model numeric types and *BooleanType* to model boolean types. In addition, it allows us to model Java-based types using *EObjectType* and enumerations using *EnumerationType*. It also provides the capability to model complex types like *ArrayType* to design array types and *RecordType* to design structured types. *Type* represents an abstract type inherited by *Mixin* and *Kind*. *Constraint* represents a business invariant related to a specific cloud domain. For example, all IP addresses of all network resources must be distinct. *MixinBase* represents an instantiation of the *Mixin* concept in an entity. *AttributeState* represents an instantiation of the *Attribute* concept. Based on the OCCIware metamodel, OCCIware Studio [13], the first model-driven tool chain for OCCI, has been developed. It provides a model-driven tooling to design and verify both OCCI extensions and configurations. In addition, it provides a set of generators to produce artifacts such as documentation and deployment scripts.

4 MoDEMO Approach

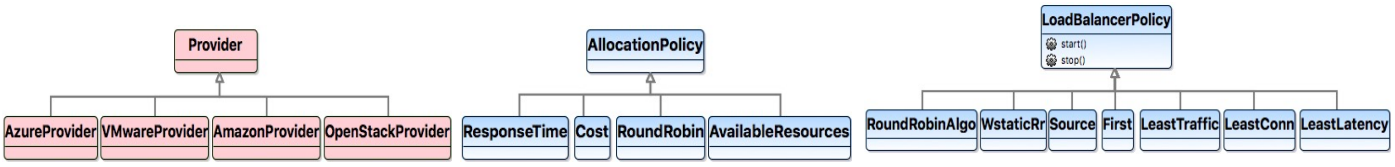
In this section, we present MoDEMO. We begin by describing the general overview. The model is then described. We also describe the different policies and the model design configuration. It is worth noting the figures in this section are directly captured from our Eclipse project.

4.1 MoDEMO general overview

MoDEMO metamodel (for simplicity model) is defined in the upper part of Fig. 2. The operators can configure different instances according to their needs, this part abstracts all the aspects of elasticity. The operators can be the cloud architects or the system administrators. The bottom part of Fig. 2 represents the multi-cloud environments. In the middle part, a connector is charged to synchronize the model abstraction and the real multi-cloud environments by executing actions, monitoring state, etc. The upper part is Models@run.time which is a reflection of the multi-cloud environments.

4.2 MoDEMO Model

From OCCI perspective, the MoDEMO model defines the following Resource kinds (red colored kinds in Fig. 3): *HorizontalGroup*, *Provider*, *LoadBalancer*, *Compute*, *ElasticityController*, *Step*, *ActionTrigger*, etc. All resource kinds support *CRUD* operations (i.e., `Create`, `Retrieve`, `Update`, and `Delete`). The Compute kind, already defined in the OCCI Infrastructure specification, is inherited by different clouds



(a) Cloud providers (b) Allocation policies (c) LB algorithms

Fig. 4: Cloud providers, allocation policies and load balancing algorithms

or compute providers. In addition, the model has seven Link kinds (green colored in Fig. 3): *ElasticLink*, *StepLink*, *Rule*, *InstanceGroupLink*, *LoadBalancerLink*, *ProviderLink*, and *ProviderInstanceLink*. There are also many *mixins* (blue colored in Fig. 3) such as *Metric*, *SchedulingPolicy*, *ScalingPolicy*, *AllocationPolicy*, *MigrationType*, etc. that can be applied to the other entities (Resource and Link instances). In the following, we present briefly the main concepts of our MODEMO model (c.f., Fig. 3 - all the figures presented in this article can be found here[2]).

4.2.1 `HorizontalGroup` (HG)

The HG represents a collection of compute instances that share similar characteristics and are treated as a logical unit for the purposes of elasticity and management, *e.g.*, an application that operates across multiple instances. The HG owns a set of `attributes` including a name (`name`), a minimum number of compute instances (`minimum`), a desired capacity or number of instances that the HG attempts to maintain (`groupSize`), a maximum size of instances (`maximum`), and a template (`templateName`) or launch configuration which specifies the image or application of the HG compute instances. In addition, the HG is linked to a load balancer (`loadBalancer`) to distribute the traffic among its compute instances.

The instances of a horizontal group can be seamlessly deployed on multiple clouds according to a certain policy. Thanks to the `AllocationPolicy` mixin that can be dynamically applied on the horizontal group.

4.2.2 `AllocationPolicy`

This *mixin* permits to choose the appropriate provider (`Provider`) in order to deploy a new horizontal group compute instance. Our model supports different policies (mixins) as shown in Fig. 4b:

- `RoundRobin`: This policy selects a provider in turns, starting with the first one from the list of providers in which the horizontal group has links to, until the end of the list is reached at which point the next request (instance provision) will go back to the first provider again.
- `ResponseTime`: The nearest provider will be chosen.
- `Cost`: This mixin will choose the low-cost cloud, the provider which offers the lowest price for deploying the instances.
- `AvailableResources`: This policy chooses the cloud which has sufficient available resources, especially the private cloud based on VMware or OpenStack.

2. https://github.com/yehia2221/MoDEMO/tree/master/figures

4.2.3 `Provider`

The `Provider` kind models a cloud provider entity. Multiple providers can be used simultaneously in order to ensure the redundancy and reliability of the system. As shown in Fig. 4a, the currently supported providers are:

- `AzureProvider`: Microsoft Azure cloud service.
- `VMwareProvider`: cloud based on VMware technology.
- `AmazonProvider`: Amazon Web Services (AWS).
- `OpenStackProvider`: cloud based on OpenStack.

4.2.4 `LoadBalancer` (LB)

The `LoadBalancer` kind represents a LB which is used to distribute traffic among the HG compute instances. The LBs are used for optimizing resource utilization, reducing latency, maximizing throughput and ensuring fault-tolerant configurations. There are some key concepts and terms associated with the LBs such as frontend, backbends, algorithms, etc. The frontend defines where and how the incoming traffic is reached to the machines behind the LB. The backend defines a pool (list) of servers called backend servers that the frontend will forward requests to. LB algorithm is used to determine which server, from the backend list, will be selected when load balancing. As shown in Fig. 3, the LB has a set of `attributes` and `actions`. The most important actions are `addbackendserver()` and `removebackendserver()` which serve to register/remove the compute instances belonged to a HG in/from the LB. In addition, our model offers a set of mixins (`LoadBalancerPolicy`) as shown in Fig. 4c which support the most utilized LB algorithms. Such mixin can be dynamically applied on the LB, each mixin has two operations, `start` (to start the algorithm) and `stop` (to stop the algorithm). The supported LB policies are:

- `RoundRobinalgo`: This mixin or Round Robin algorithm selects servers in turns, starting with the first one in a backend until the end of the list is reached at which point the next request will go back to the first server again. The servers list in the backend may be assigned a weight parameter to determine how frequently the server is selected. Therefore, using Round Robin algorithm, each horizontal group compute instance is used in turns according to their weights.
- `WstaticRr`: This mixin policy is a representation of Static Round Robin algorithm. Similar to the previous algorithm where each server is used in turns, according to their weights. However, it is static, which means that changing a server's weight on the fly will have no effect.
- `Source`: This mixin selects which server to use based on a hash of the source IP. The source IP address is hashed

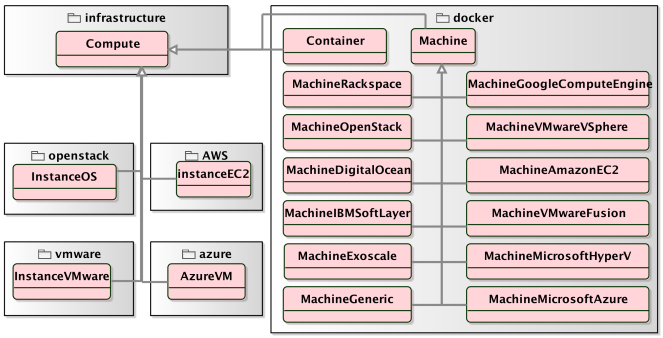


Fig. 5: Compute instances

and divided by the total weight of the running servers to designate which server will receive the request.

- `First`: The first server with available connection slots receives the connection.
- `LeastConn`: Each resource (compute instance) in a given backend is evaluated to determine which one has the least number of active connections. The next request will be forwarded to the server which has the least number of active connections.
- `LeastTraffic`: The request will be forwarded to the server which has the least outgoing traffic.
- `LeastLatency`: The request will be forwarded to the server which has the least latency.

4.2.5 `Compute`

The `Compute` kind abstracts computing resources that can be VMware instance (InstanceVMware), OpenStack (oscore instance), AWS EC2 (instanceEC2), GCP instance or Container instance, etc. As shown in Fig. 3, this part extends the OCCI **Infrastructure** [17] model, which abstracts Cloud infrastructure resources (i.e, Compute, Network and Storage). The Compute has a set of `attributes` and `actions` including *CRUD* operations. These attributes and actions determine the characteristics and behavior of a Compute instance. It is worth noting that there is a specific model for each technology which inherits from the Compute. Regardless of the complexity that hides in the representation of VM and container instances, our model deals with them as Compute instance with generic methods. This will facilitate the addition of a new specific provider's infrastructure. Fig. 5 is only a simplified version of the model, a more detailed capture figure taken from Eclipse IDE can be found here[2].

4.2.6 `ElasticController` (EC)

The `ElasticController` kind represents the elasticity controller. The EC is the core component of this model. As shown in Fig. 3, many mixins or policies can be applied on the elastic controller. We describe these policies in Section 4.2. In addition, the relationship between the EC and the other components will be highlighted in the coming paragraphs. EC has some attributes such as `minimumLimit`, `maximumLimit` which determine the minimum/maximum limit the EC can scale the resource. The resource here is not OCCI resource; it is HG, VM, Memory, vCPUs, etc. EC has also a constraint (stepconstraint) that allows `Steps` to be only used when the EC has a `StepDynamicScalingPolicy`.

4.2.7 `Step`

The `Step` kind helps the EC to choose the amount/number of resources/instances to be increased/decreased. Based on the *step bounds* and `size` attributes, EC will decide the amount or size of resources to be increased or decreased. Step works with the `StepDynamicScalingPolicy`. Its utility will be described in Section 4.3.

4.2.8 `ActionTrigger` (AT)

AT is used by the EC to perform elasticity actions. AT has three attributes (`action`, `actionType`, `amount`) as shown in Fig. 3. The `action` specifies the action operation that can be "*add*", "*remove*" or "*set to*". `ActionType` can be "*percentage*" or "*instance*". `amount` determines the amount or size of the added/removed resources, it is based on the `ActionType` attribute to determine its value.

4.2.9 `Rule`

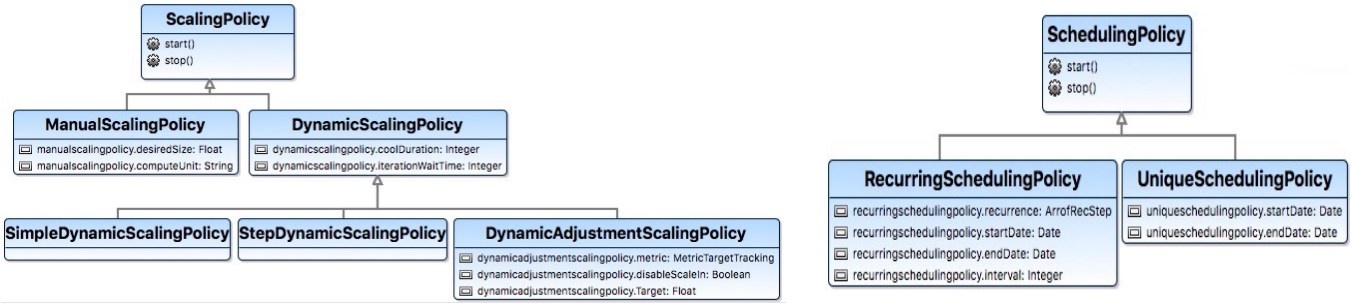
The `Rule` kind links an EC to an action trigger, each EC has one or many rules. Each rule link is associated with one `ActionTrigger` (target). A rule is used to evaluate certain metric. The metrics can be CPU, Memory, Response Time that are represented in a form of *mixins* which can be added/removed dynamically to the Rule (`Metric` mixin in Fig. 3). A Rule owns a set of attributes such as `operator` (equality or relational operator), `threshold`. In addition, `period` and `consecutive` attributes will be passed to the monitoring system to determine the period and interval for the metrics to be monitored. A rule has a constraint (ruleconstraint) to restrict a rule to only have one mixin (metric) at a time.

4.2.10 Other links

In OCCI, link is a relation between two resource instances. A link references both source and target resource, *e.g.*, when EC is linked to HG, information about the target (HG) can be sent to the source (EC). OCCI requires *links* to connect the different related resources. As shown in Fig. 3, our model has the following links. `LoadBalancerLink` links a HG to a LB. Each HG has one LB. `InstanceGroupLink` links a HG to a Compute instance. HG has one or many Compute instances. `ElasticLink` links an EC to a HG (in case of horizontal elasticity) or a Compute instance (in case of vertical elasticity). `StepLink` links an EC to a step. EC may have zero or many steps. `ProviderLink` links a HG to a provider. The compute instances of a horizontal group can be deployed on one or many providers. `ProviderInstanceLink` links Provider to Compute. This link distinguishes which Compute instances belong to which cloud provider. A provider can have zero or many compute instances.

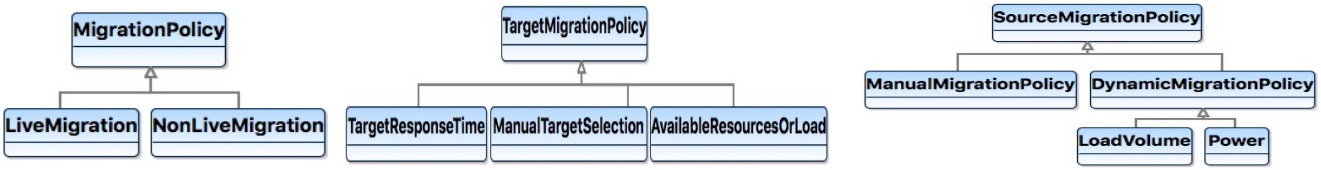
4.3 MoDEMO policies

As shown in Fig. 3, an elastic controller is enabled with different types of policies or strategies including scaling and scheduling policies, swapping and migration. Multiple elasticity policies supported by the most popular cloud



(a) Scaling policies (b) Scheduling policies

Fig. 6: Scaling and scheduling policies



(a) Migration types (b) Target migration policies (c) Source Migration polcicies

Fig. 7: Migration

providers and more have been considered in MODEMO. In order to support the different policies offered by other cloud providers, our analysis procedure takes different steps: *i*) reading the cloud provider documents and the corresponding API support. *ii*)- testing the policy in practice by using the provider web console. *iii*)- proposing UML class diagram model for each provider in abstracting the different aspects related to elasticity, *iv*)- finding the best compromise between the different policies in order to integrate them in MoDEMO system. There are four different types of scaling policies (Fig. 6a). In addition, there are two scheduling policies that can be applied to the scaling policies in order to run them in the future (Fig. 6b). Migration and swapping are also complementary policies that indirectly enhances the elasticity concept. Therefore, our EC has more than fourteen different modes that enable the adjustment of resources. We introduce these policies in the subsequent subsections:

4.3.1 Scaling policies

Scaling policies are elasticity policies which trigger elastic actions as well as define how the system adapts to workload and resource utilization. As shown in Fig. 6a, there are four scaling policies:

A- `ManualScalingPolicy` (MSP)
It means that the user is responsible for performing elasticity actions based on her/his choice. MODEMO provides an interface to allow performing vertical or horizontal elasticity actions on the HG or compute instances.

B- `SimpleDynamicScalingPolicy` (SimpleDSP)
As other scaling policies, *SimpleDSP* permits to execute elasticity actions dynamically. However, when and how to scale in/out or up/down resources differs from one policy to another depending on the policy type and the algorithm used. As shown in Fig. 3, the EC may have many rules and each rule is associated with its `ActionTrigger`. As explained before, the rule evaluates a metric, *e.g.*, if the CPU utilization (`Metric`) during an interval of 2 minutes (`periods`) for 3 consecutive periods (`consecutive`) is greater than (`operator`) 90 (`threshold`), then the simple dynamic policy may execute the linked action to that rule. The simple dynamic policy will classify the rules into two categories: ***i)*** rules that have an increase action and ***ii)*** rules that have a decrease action. For the first category, it will evaluate (logical *or*) between the rules and will execute the first action of the first rule that is true. For the second category, the rules will be evaluated by a logical *and*, if all the rules are true in this category, a random action will be chosen from the actions associated with these rules.

C- `StepDynamicScalingPolicy` (StepDSP)
The main difference between this policy and the previous one is the size or amount of the added/removed resources (VM instances, CPUs, memory) per scaling action. In this policy, the size is determined by the metric usage, threshold and step limits. If the EC has no steps (`Step`) associated with it, StepDSP will simply execute the action associated with the rule when the rule is true. If steps are associated with the EC, the metric usage will be compared with $(threshold \pm lowerStepBound)$ and $(threshold \pm UpperStepBound)$ and will take the *size* attribute in Step as the amount or number of resources to be added/removed instead of the amount defined in `ActionTrigger` as shown in Fig. 3. To sum up, the metric utilization is compared with the ranges specified in the Steps to determine the number/amount of resources to be added or removed.

D- `DynamicAdjustmentScalingPolicy` (DASP)
DASP or target tracking scaling policy performs dynamic scaling based on a target value. A target value or threshold is set for a certain metric. The DASP will keep monitoring

the metric and trigger scaling policy actions to increase or decrease resources based on the target value. The scaling policy adds or removes capacity as required to keep the metric at, or close to, the specified target value. However, the scaling down actions are restricted to the formula: $\left(\frac{(count-1)*T_{metric}}{count}\right) - \alpha$, in order to avoid the rapid oscillation of scaling in/out, down/up actions [18]. T_{metric} is the target value (threshold), $count$ is the number of current compute instances, vCPUs or memory size depending on the elasticity type, α is a small integer. The metric usage must be less than the value returned by the above formula.

4.3.2 Scheduling policies

We present the scheduling policies as shown in Fig. 6b. Scheduling policies are policies which trigger elasticity actions and scale the application in response to predictable load changes in the future.

A- `UniqueSchedulingPolicy` (USP)
An EC can execute elasticity decisions based on a schedule in response to predictable load changes. USP can be applied to the MSP, SimpleDSP, StepDSP, and DASP in order to be fired/stopped at the start/end dates specified. It gives also the possibility to cancel the scheduled future actions.

B- `RecurringSchedulingPolicy` (RSP)
Similar to USP, RSP can be applied to the MSP, SimpleDSP, StepDSP, and DASP in order to be fired in the future. However, recurring scheduler allows the action to be repeated in the future at the time specified. RSP can start/stop the recurrence based on start/end dates specified, however the dynamic policies (DP) can run infinitely, therefore RSP gives a control on DP. For example, RSP can fire the SimpleDSP every weekend for ten hours. RSP accepts all the *cron expressions* [19], which are able to create firing schedules based on the expression specified. For example, At 8:00 am every Monday through Friday or At 1:30 am every last Friday of the month, etc.

4.3.3 `SwappingPolicy`

The above elasticity policies are considered as the de facto solution to support the timely provisioning and de-provisioning of resources, this elasticity can still be broken by budget requirements or physical limitations of a private cloud. SwappingPolicy is an alternative, yet complementary, solution to the problem of resource provisioning by adopting the principles of swapping in the context of cloud computing. In particular, this policy consists in detecting idle virtual machines, containers to recycle their resources when the cloud infrastructure reaches its limits. Cloud providers use the technique of overcomitting (*e.g.*, over-subscription in VMware, overcommitment in OpenStack) in order to virtually increase their capacity. Overcommitting is a technique of allocating more virtualized CPUs or memory than the real physical resources. These techniques allow to deploy more workloads but stability issues can occur and major performance problems can be introduced affecting all of the workloads running on the infrastructure. In [20], we evaluated the impact of resource overcommitment on VM performance, and we found that once the number of allocated physical core is reached on a compute node (P(CPU)), the performance becomes linearly impacted by the provisioning of new VMs. Additionally, in [21], they found that the resource oversubscription with ratios 1:1 to 3:1 has no problem on performance but 3:1 to 5:1 may begin to cause performance degradation. This policy will recycle the unused resources and will automatically restore them on demand via listening through a proxy for their requests.

4.3.4 `MigrationPolicy`

Migration can be also considered as a needed action to further allow the elasticity when there is no enough resources on the host machine. However, it is also used for other purposes such as migrating a VM to a less loaded physical machine just to guarantee its performance or to another destination in order to power off the source host and save energy, etc. Migration types are categorized as live or non-live VM migration as shown in Fig. 7a. A live migration does not suspend application service during VM/container migration, whereas a non-live pattern follows pause, copy, and resume methodology to migrate a VM/container. In our model, the migration policy depends on `TargetMigrationPolicy` to choose the destination to where the migration will be.

4.3.5 `TargetMigrationPolicy`

The EC will use this policy to choose the target provider or data center that the VM/container will be migrated to. Our model supports three mixins that choose the target destination for the migrated workloads as shown in Fig. 7b.

- `ManualTargetSelection`: The user will choose his/her preferred provider/data center manually.
- `TargetResponseTime`: The nearest target will be chosen.
- `AvailableResourcesOrLoad`: The target with most available resources will be chosen.

4.3.6 `SourceMigrationPolicy`

Similar to the other policies, MODEMO offers a mean for migration based on the EC `Rule` and its associated `Metric`. However, since the migration has various objectives such as power management, fault tolerance, load balancing, system maintenance, performance issues, etc., `SourceMigrationPolicy` permits to direct the migration process based on certain objective. If this mixin is not applied, the EC will use its rules and metrics to decide when to trigger the migration process. If `SourceMigrationPolicy` is applied, it will direct the migration process based on certian objective, *e.g.,* saving energy. It has many mixins where each mixin represents an algorithm based on a single objective as shown in Fig. 7c.

- `ManualMigrationPolicy`: Based on the user choice to migrate, *e.g.*, if the user decides to effectuate maintenance, this mixin can be used.
- `Power`: The migration will be triggered in order to save energy.
- `LoadVolume`: The migration will be fired when an overload is detected, *e.g.*, storage is running out of space.

In general, `MigrationPolicy` specifies the migration method, `TargetMigrationPolicy` determines

where to migrate, the EC `Rule` and `Metric` or `SourceMigrationPolicy` determines when to migrate. If EC is connected to a single Compute, that Compute will be the entity to be migrated but if the EC is linked to a HG, one or all of its instances will be migrated.

4.4 MoDEMO design tool

MODEMO is implemented as a set of Eclipse plug-ins on top of OCCIware studio [13]. Fig. 9 shows a runtime configuration of our MODEMO model. All the parameters in the EC and its associated rules, steps, metrics, action triggers can be modified at runtime as shown in Fig. 9 and indicated in Section 2. MODEMO is powered with plug in/out functionalities. As shown in Fig. 9, we drag and drop any policy from the design tool to the EC configuration which eases the utilization of our framework. The metrics that are evaluated by the rules can be also dragged in/out dynamically in the Rule box as shown in the configuration, thanks to the OCCI *mixins* which permits adding/removing plugins dynamically to the system. A list of the configurable settings at runtime will be presented in Section 5.

5 VALIDATION

5.1 Experimental setup

MODEMO has a wide range of components, therefore we evaluated some important aspects of our work by using Flask application version 0.10.1 [22] and Redis database (DB) version 3.2.0 [23]. Flask is a framework for Python based on Werkzeug which computes a calculation (through REST API) and stores the results into the database (Redis). Flask runs an application for Fibonacci numbers, written in Python. The reason we use Fibonacci with flask application is that it intensively consumes resources (CPU, memory). In addition, each operation takes a long time to be calculated. Apache HTTP server benchmarking tool (ab) is used to generate workloads to the Flask application in order to calculate the mathematical operations (Fibonacci sequence). The result of the mathematical operation is stored in the Redis DB. We performed all our experiments on Scalair, AWS and MS Azure infrastructures. AWS and MS Azure as described before are the well-known cloud providers. Scalair is a private cloud provider company. Scalair infrastructure is managed by VMware vSphere 6.0 technology. The hardware specifications consist of 2 powerful servers: 2 HP ProLiant DL380 G7 (Intel(R) Xeon(R) CPU X5650 @ 2.67GHz, 84 GB, 6 NICS).

5.2 MoDEMO implementation

Fig. 8 gives a high-level illustration of MODEMO implementation to understand the concepts behind our architecture. This architecture is composed of the following parts: MODEMO model manager, Provider connectors (OCCI) (*e.g.*, VMware connector), Zabbix monitoring system, and the load balancer HAproxy. MODEMO model and its connector are described at the beginning of Section 4. Fig. 8 defines the relationship between the different components of the system. MODEMO manager communicates with the provider-specific connectors which in turn will provision, execute actions on the executing environment via their specific orchestrators, *e.g.*, Vcenter, Docker machine/engine, etc. In addition, MODEMO must capture the resource utilization in order to allow the elasticity controller to perform scaling actions according to the chosen policy. Therefore, MODEMO communicates with our monitoring system (Zabbix). Zabbix is an open source monitoring tool that works with a centralized Linux-based Zabbix server [24]. According to the type of elasticity, MODEMO must register/remove the instance (VMs) in a load balancer. We use HAproxy which is fast and reliable open source solution offering high availability, load balancing, for implementing complex load balancing in terms of different algorithms for traffic distribution [25]. In addition, if a compute instance is removed, it will be deregistered from both the monitoring system and the load balancer. It is worth noting that MODEMO uses generic methods, for example, it deals with a compute instance rather than VM instance directly.

5.3 Research questions

Our evaluation answers the following questions:

- *Q#1*: Are all the AWS, MS Azure and GCP scaling policies supported by MODEMO?
- Q#2: What are the possible configurations or settings at runtime?
- Q#3: What is the overhead introduced by our MODEMO model?

MODEMO is a unified framework that enables all the elasticity policies supported by the worldwide cloud providers. We have chosen AWS, MS Azure and GCP as indicated in *Q#1* because they are the major actors in the cloud market. According to Synergy Research Group (SRG) [26], AWS, MS Azure and GCP represent the majority of cloud market share in the world, *e.g.*, Amazon maintained its dominance as its market share to 34% in the second-quarter of 2018. Secondly, MODEMO supports runtime configurations, therefore *Q#2* will verify what are the different components that can be configured at runtime and how safely these components can be plugged in/out at runtime. Thirdly, since our system is configurable at runtime, it could impact the performance and that is why we measure the overhead in *Q#3*.

Q#1. In order to answer this question, we compare MODEMO elasticity policies and the corresponding AWS, MS Azure, GCP scaling policies in Table 1. Our comparison is based on the analysis procedure described in section 4.3. However, this time, a much more concentration is given on the provider-specific web console in order to test each policy individually in practice and to compare it with MODEMO policies set in a runtime configuration instance.

Table 1 shows that MODEMO supports all the elasticity policies found in AWS, MS Azure, and GCP. However, MODEMO not only supports the elasticity policies in the above popular providers, it suggests some improvements. For instance, simple policy in AWS only supports one metric (rule) per policy, MODEMO permits many rules to be aggregated in the same policy. The schedulers in MS Azure are limited to certain dates and days while MODEMO supports any combination and recurrence frequency. In addition, MODEMO avoids the rapid oscillations as explained in the DASP because the scaling down actions are not only limited to the target usage but also to the group size. Finally,

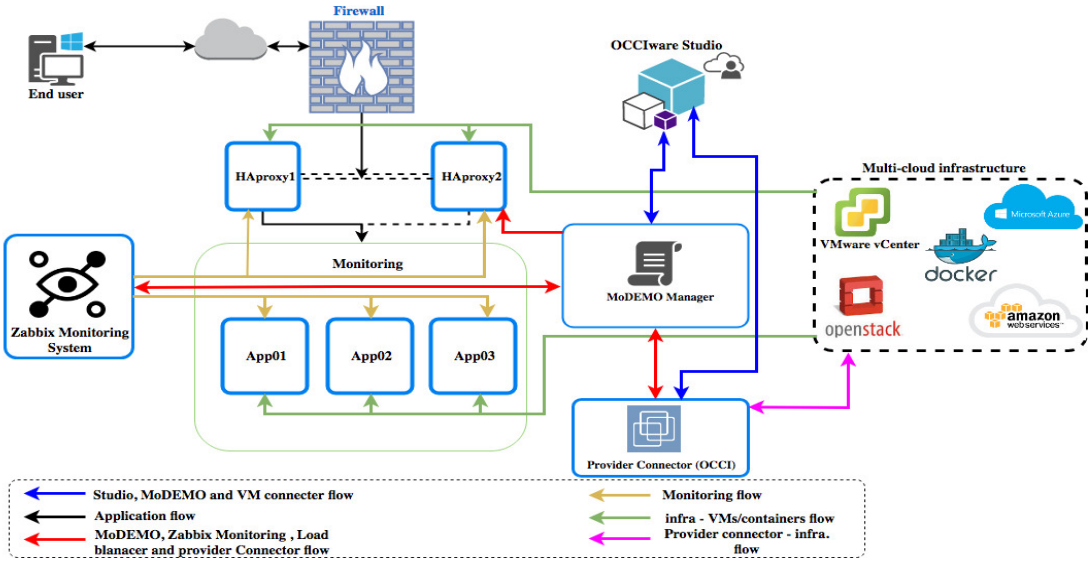


Fig. 8: Architecture overview

TABLE 1: Elasticity Policies Supported by Public Cloud Providers and MoDEMO

MoDEMO	AWS	Azure	GCP
MSP	Manual	Instance account	Manual
MSP with USP	Scheduled	Scheduled	–
MSP with RSP	Scheduled cron	Scheduled	–
SimpleDSP	Simple Policy	Metric based	–
SimpleDSP with USP	–	Scheduled (specific dates)	–
SimpleDSP with RSP	–	Scheduled (specific days)	–
StepDSP	Step Policy	–	–
StepDSP with USP	–	–	–
StepDSP with RSP	–	–	–
DASP	Target tracking	–	autoscaler (single/multiple metrics)
DASP with USP	–	–	–
DASP with RSP	–	–	–
Swapping	–	–	–
Migration	AWS Server Mig. Service	Windows-built mig.	CloudEndure

these policies are applicable for both vertical and horizontal elasticities, while the supported policies in AWS and GCP can only be applied to the horizontal elasticity.

Q#2. In this section, we have deployed our application as described before, where MoDEMO used to deploy and control the elasticity. Fig. 9 is a MoDEMO runtime configuration for the deployed application. It shows some of the components of the model. Frame (a) in Fig. 9 shows the Eclipse Model Explorer used to navigate through the different project configurations containing our MoDEMO Model. Frame (b) displays the design area that provides a graphical representation of some components of MoDEMO Model. Frame (c) in Fig. 9 displays the configuration pallet that represents the MoDEMO Model elements such as HG, link, LB, Compute, etc. Frame (d) in Fig. 9 gives an outline or a global view of the modeled components (EC, application, infrastructure, LB, etc.). The Eclipse properties editor for visualizing and modifying attributes of a selected modeling element, console tab, errors tab, etc. are at the bottom of the figure, we closed them to gain space.

At the beginning of the experiment, the HG (Scalair HG in Figure 9) has only one VMware VM instance (Flask1). With the increased numbers (Fibonacci numbers) send to the application, other two instances (Flask2 and Flask3) are provisioned on AWS and Microsoft Azure respectively. The instances are provisioned subsequently on the different providers. This is because of the allocation policy (*RoundRobin*) set on the HG which is used to choose the cloud provider. If there is other more instances to be provisioned, the next instance will be deployed on the private VMware provider and etc. as shown in Figure 9. During the workload generation, MoDEMO adds more instances, at the same time we also change many parameters and components at runtime. Our aim of this experiment is not only to verify which settings can be changed at runtime but also to verify how safely different components of the system can be added/removed. For example, the different providers are added safely at runtime and our system continues to monitor and distribute the traffic among the different instances normally. In the following, a list of components and parameters that can be changed at runtime.

- Cloud providers
- Allocation policies
- Elasticity policies
- Metrics
- Elasticity controller parameters
- Rules, action triggers
- Horizontal group parameters
- Load balancer policies and parameters

During the execution of the experiment, we change different values and components as shown in Fig. 9 at runtime. The above items lists some of the important parameters and components that control the behavior of elasticity while

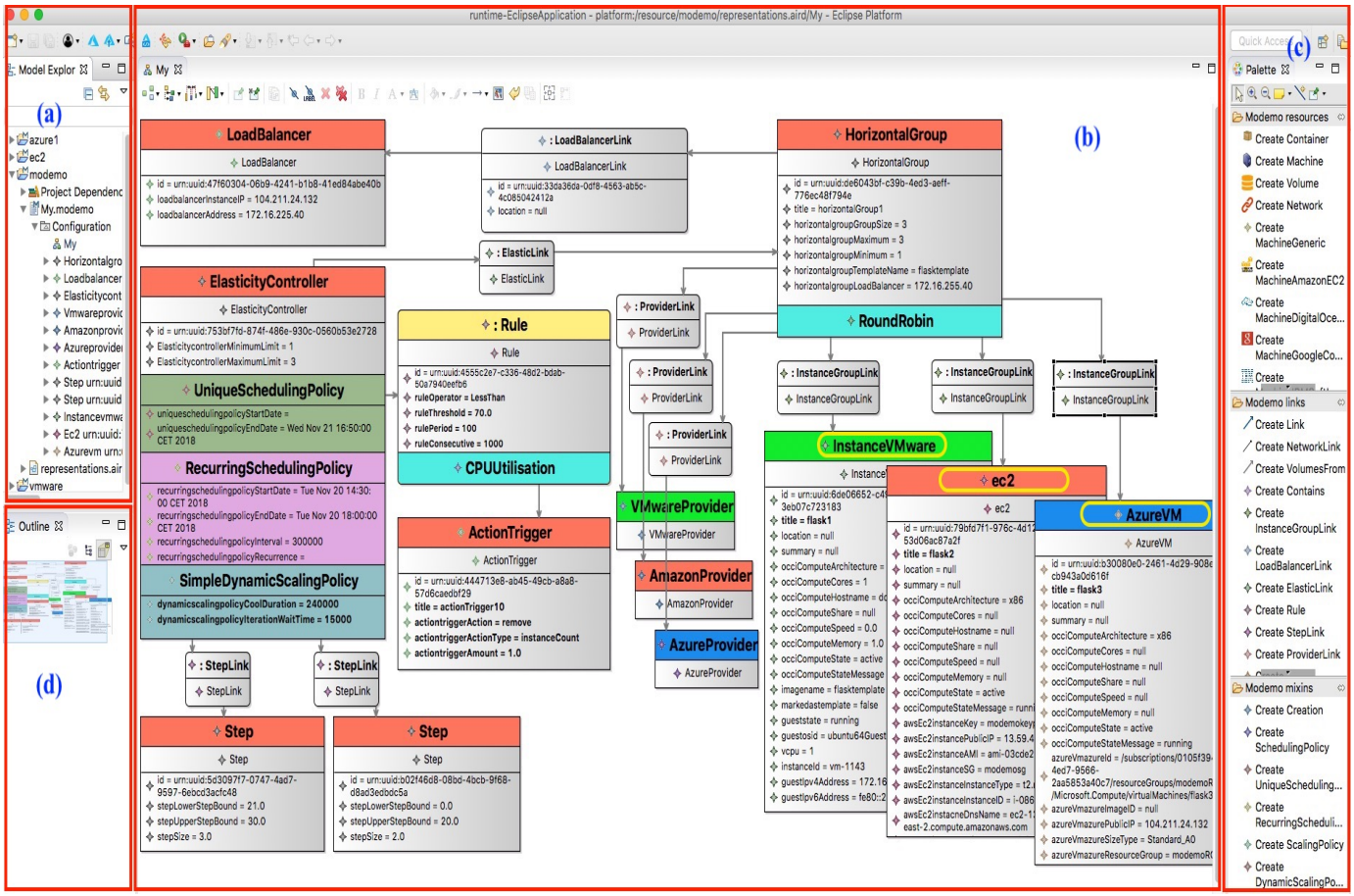


Fig. 9: MoDEMO Runtime Configuration

there are other parameters and mixins such as *name*, *title*, *load balancer algorithm mixins*, etc. can be changed at runtime. In addition, components of the model such as *Rules* and *AT* can be added dynamically to the design at runtime when the *SimpleDSP* is applied. To integrate elasticity controller with a new scaling policy, it is enough to drag it from the design tool. With the possibility to drag/drop policies/metrics to the configuration as shown in Fig. 9, MODEMO eases the utilization for users. The Compute instances and the LB are deployed and configured transparently and independently of any deployment tool.

Q#3. To determine the overhead introduced by our MODEMO model, we evaluate two scenarios where our application described previously in this section is deployed. A certain workload is generated to the application. The application is deployed and the elasticity is achieved using two scenarios: i) with a script which runs directly on the infrastructure, the scripts used in this experiment can be found here[3], and ii) with MODEMO model. Of course, we take two types of the elasticity policies which are the simpleDSP and the StepDSP. The two scenarios are repeated ten times for both the vertical and horizontal elasticity, the average values are reported.

1) Vertical elasticity: We measured the average total time it takes to execute the workload and perform elasticity actions, i.e. reconfiguration of the VM resources. In Table 2, we present the results of the average total time for executing the workload using the vertical elasticity for both scenarios (script runs directly on the infrastructure and MODEMO), as well as the median, minimum and maximum values, and the overhead introduced by the MODEMO Model. The overhead introduced by our model is **1.1%**. It is worth noting that during the experimentation of the two scenarios, we have set the same values of the elasticity parameters such as *threshold*, *cool duration*, etc. for both types of policies in the two scenarios.

TABLE 2: Vertical elasticity overhead

Scenario	Avg. total time	Median	Max. value	Min. value	Overhead
Script on the infr.	392.9323 sec	392.3085	398.697	387.629	-
MoDEMO Model	397.2353 sec	396.281	406, 871	392.941	1.1%

2) Horizontal elasticity: Similarly, we have generated a workload to our application, we measured the time it takes to add more instances and to execute the workload in the context of the horizontal elasticity. Table 3 shows the average total time that it takes to achieve the horizontal elasticity and manipulate the workload (the scientific calculations) in the two scenarios, in addition to the median, maximum and minimum values, and overhead. The overhead introduced by our model is **1.95%**. The overhead with the horizontal elasticity is a little bit bigger than the vertical elasticity, the reason is that simply the horizontal elasticity involved with more model configurations and components. The median for both types of elasticity is very close to the mean total execution time which indicates that there are no outliers that

3. https://github.com/yehia2221/MoDEMO

are much greater or smaller than most of the other values of our ten experiments.

TABLE 3: Horizontal elasticity overhead

Scenario	Avg. total time	Median	Max. value	Min. value	Overhead
Script on the infr.	414.0541 sec	413.7725	421.442	404.607	-
MoDEMO model	422.1385 sec	421.782	427.802	416.485	1.98%

These experiments show that there is just a very small overhead introduced by adding the MODEMO model. The overhead is negligible regarding all the advantages provided by our approach.

6 DISCUSSION

MODEMO is a unified system that supports vertical and horizontal elasticity for both VMs and containers along with seamless and simultaneous use of multiple cloud infrastructures. It is based on a standard representation of the infrastructure entities and elasticity components. This system is demonstrated on real platforms using real application. Our extension is modular, new policy or resource type such as a cloud provider can be added without changing the model. This is generally as a result of the model design and OCCI standard where all extensions respect a common paradigm. For example, VMs and containers from any providers are represented as the same compute units from the elasticity controller perspective, and it use general methods, *e.g.*, *occi-Create*, to manage these resources transparently. However, it applies the elasticity policies from the already defined ones, but if the provider has a new policy, it should be defined in the list of plugins to the elasticity controller. The elasticity controller uses the metric type to determine which resource (e.g., CPU or memory in the vertical elasticity) to add or remove. MODEMO is a unified framework that supports different clouds and different forms of resources. There are two main scenarios in which MODEMO has a great utility. First, when multiple clouds are used, it is better to use a single framework to manage these providers where MODEMO adds more features that are not supported by a single provider. Secondly, when the user needs to use different policies rather than those supported by the cloud provider itself. However, if the user uses only a single cloud provider with its supported policies, apart from the friendly MODEMO user interface, it is the same to use either MODEMO or the dedicated cloud interface. Although MODEMO permits to execute several elasticity policies, it has some limits. MODEMO offers the possibility to enable different policies, however, there is no synchronization between them. It allows launching different strategies but each one is independent of the other. This might negatively impact the system by providing more resources if two policies are fired simultaneously. Depending on the policy, its corresponding algorithm and defined thresholds, concurrent policies might result in contradicting decisions. We plan to overcome this limit by implementing an intelligent system that keeps a track of the different triggering policies and analyzes their behavior when adjusting the resources. The user must also pay attention for the settings, *e.g.*, lower/upper bounds of the step policy. It is the responsibility of the user to not let a gap between the different steps or to not make overlapping of the threshold values. Additionally, the algorithms are rule-based, while they can be adapted and changed at runtime, MODEMO could be improved to integrate predictive and machine learning approaches along with the reactive strategies.

7 RELATED WORK

To the best of our knowledge, no approach has been proposed so far on managing elasticity and unifying its different policies that are driven by the models in the cloud. We, therefore, present and discuss the closest related works that are relevant to our approach. Gandhi et al. [27] proposed cloud service, Dependable Compute Cloud (DC2), that dynamically scales the application deployment based on user-specific performance requirements. This approach makes use of a queuing network model and Kalman filtering technique to determine the necessary scaling actions. CloudMIG [28] is a model-based approach for migrating legacy software systems to scalable and resource-efficient cloud-based applications. The solution model focuses on reengineering activities during the migration of existing software systems to scalable and resource-efficient Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) rather than a representation of elasticity policies or infrastructure entities. Ali-Eldin et al. [29] proposed hybrid horizontal elasticity controller that incorporates both reactive and proactive components to dynamically change the number of VMs allocated to a service running in the cloud based on the current and the predicted future demand. This approach models the state of the system, e.g., infrastructure current servers, number of requests, etc. and the future usage but the elasticity controller is a single entity. [11] describes a Multi-Cloud-PaaS (MCP) architecture to manage elasticity across multiple cloud providers. In this work, they present a non-modular, flat architecture to deploy an application on a list of static catalog of cloud providers. The mentioned proposals, as most of other works around elasticity, represents the elasticity controller as a single black box entity while our work permits to dynamically add/remove different components, thanks to the dynamic modular nature of the design and method used.

In [30], the authors proposed an approach based on a model to dynamically add autonomic facilities to cloud resources. Compared to our model, this work only tackles the problem of elasticity at SaaS level. It does not provide clear isolation between the managed resources and the elasticity components. It also does not deal with heterogeneous environments and mechanisms such as different elasticity types, cloud providers, virtualization technologies. Elasticity actions are limited to three configurations and static thresholds. rSYBL [31] is a framework for multi-level cloud service elasticity control, considering requirements associated with multiple abstraction levels. The multi-level concept in this proposal refers to the cloud services (applications) and cloud resources (infrastructures). The interesting part of this work is the use of the standard TOSCA to represent the applications, mapping between them and their deployment on the infrastructure. However, the elasticity itself remains a highly theoretical representation which we think it is a far away from a real implementation. MODEMO abstracts each aspect of elasticity and provides a clear

and real links between the different resource abstractions. SRL [32] is a scalability rule language, a language for specifying scalability rules that support cloud applications. In comparison with MODEMO, SRL is limited by design to specific scaling policies. Additionally, MODEMO objectives are to provide unification and to release users from interacting directly with infrastructures, so everything is abstracted. SRL does not provide a complete abstraction for different cloud providers and different computing units. STARTModel [33] is descriptive domain specific language for expressing elasticity model for business processes. The elasticity controller generated from this model is based on a predefined template where the scaling policy and even elasticity actions, system transitions are statically defined in a predefined space. This work is also user-oriented, it tends to meet more the service user perspectives rather than provider perspectives. A recent work [12] around elasticity proposes a Cloud Resource Description Model (cRDM) based on a state machine for describing the cloud elasticity. Authors in [12] defined fixed set of states for the cloud resource or application where the elasticity events and transitions loop inside the limited space. This system requires an intensive manual intervention to define the state of the system and its associated transition and events. cRDM supports multiple clouds in a condition that the provider is manually defined in the state transition. Most the works in this field are not based on dynamic modular approaches. Generally, they rely on a single metric and one elasticity policy. MODEMO is a unified, modular-based framework that covers various approaches for elasticity and it allows the choice according to the need and behavior of the system. MODEMO allows to seamlessly manage elasticity across multiple clouds and it releases the users to be aware of different low-level cloud service APIs to describe and control the elasticity of cloud services. MODEMO supports multiple elasticity policies and actions, different elasticity types (horizontal and vertical), different virtualization techniques (VMs and containers) and multiple clouds. MODEMO is a provider-oriented domain-specific language (DSL), it is a middleware for managing elasticity in multi-cloud environments (provider perspective). While the described works in this section are user-oriented DSLs, describing the service user needs or application needs in terms of elasticity (user perspective).

8 Conclusion & Perspectives

This article presented our approach named MODEMO for the Model-Driven Elasticity Management of Cloud with OCCI. MODEMO has the following features: *i*) evolutive modular design based on OCCI standard, *ii*) it addresses the heterogeneity of elasticity policies and combines the features of three popular cloud providers (AWS, Azure, GCP), *iii*) it permits reconfiguration and setting at runtime, *iv*) it provides a seamless integration of the infrastructure of different cloud providers for both computing units (VMs and containers), *v*) and easy to use with the drag and drop functionalities. MODEMO is loosely coupled unified modular system which makes it a step forward towards a standalone approach of cloud elasticity management. Among its benefits, MODEMO solves the problem of interoperability by seamlessly leasing resources from multiple clouds and the problem of heterogeneity by using different computation capabilities and various elastic strategies. Experiments demonstrate that MODEMO covers the elasticity policies provided by the well-known public providers and more, is configurable at runtime, and is with negligible overhead. Our future work comprises the integration of following aspects: *i*) enhancing our model with a predictive approach to forecast workloads and react in advance, and *ii*) proposing an intelligent coordination between the different elasticity policies. Furthermore, to verify the ease of use feature in MODEMO, we plan to ask a number of professional users in Scalair using a set of use cases based on "user manual".

Acknowledgment

This work is developed within the context of CIRRUS, a joint team between Inria and Scalair company. It is a follow-up of the OCCIware research and development project. Likewise, this work is also funded by Scalair company.

Availability

Readers can find the source code and model at *https://github.com/occiware/MoDEMO*. The figures and experimental configuration files can be found at *https://github.com/yehia2221/MoDEMO*.

References

[1] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Elasticity in Cloud Computing: State of the Art and Research Challenges," *IEEE Transactions on Services Computing*, vol. 11, no. 2, pp. 430–447, March 2018.

[2] E. B. Lakew, C. Klein, F. Hernandez-Rodriguez, and E. Elmroth, "Towards Faster Response Time Models for Vertical Elasticity," in *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, ser. UCC '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 560–565.

[3] S. Farokhi, E. B. Lakew, C. Klein, I. Brandic, and E. Elmroth, "Coordinating CPU and Memory Elasticity Controllers to Meet Service Response Time Constraints," in *2015 International Conference on Cloud and Autonomic Computing (ICCAC)*, Sept 2015, pp. 69–80.

[4] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Autonomic Vertical Elasticity of Docker Containers with ElasticDocker," in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, June 2017, pp. 472–479.

[5] Y. Al-Dhuraibi, F. Zalila, N. B. Djarallah, and P. Merle, "Coordinating Vertical Elasticity of both Containers and Virtual Machines," in *8th International Conference on Cloud Computing and Services Science - CLOSER 2018*, Funchal, Madeira, Portugal, Mar. 2018.

[6] S. Sotiriadis, N. Bessis, C. Amza, and R. Buyya, "Vertical and horizontal elasticity for dynamic virtual machine reconfiguration," *IEEE Trans. on Services Computing*, vol. PP, no. 99, pp. 1–1, 2016.

[7] Baresi, Luciano and Guinea, Sam and Leva, Alberto and Quattrocchi, Giovanni, "A Discrete-time Feedback Controller for Containerized Cloud Applications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 217–228.

[8] G. Blair, N. Bencomo, and R. B. France, "Models@ run.time," *Computer*, vol. 42, no. 10, pp. 22–27, Oct 2009.

[9] "Open Cloud Computing Interface Open Standard," http://occi-wg.org, accessed: 2017-12-21.

[10] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, no. 2, pp. 25–31, Feb. 2006. [Online]. Available: https://doi.org/10.1109/MC.2006.58

[11] F. Paraiso, P. Merle, and L. Seinturier, "Managing Elasticity Across Multiple Cloud Providers," in *Proceedings of the 2013 International Workshop on Multi-cloud Applications and Federated Clouds*, ser. MultiCloud '13. New York, NY, USA: ACM, 2013, pp. 53–60.

[12] B. Hayet, A. Mtibaa, W. Gaaloul, and B. Benatallah, "Model-Driven Elasticity for Cloud Resources," in *2018 International Conference on Advanced Information Systems Engineering (CAiSE'2018)*, June 2018, to appear.
[13] F. Zalila, S. Challita, and P. Merle, "A Model-Driven Tool Chain for OCCI," in *25th International Conference on Cooperative Information Systems (CoopIS)*, Rhodes, Greece, Oct. 2017.
[14] R. Nyrén, A. Edmonds, A. Papaspyrou, T. Metsch, and B. Parák, "Open Cloud Computing Interface - Core," September 2016.
[15] P. Merle, O. Barais, J. Parpaillon, N. Plouzeau, and S. Tata, "A Precise Metamodel for Open Cloud Computing Interface," in *8th IEEE International Conference on Cloud Computing (CLOUD 2015)*. New York, United States: IEEE, Jun. 2015, pp. 852 – 859.
[16] J. Parpaillon, P. Merle, O. Barais, M. Dutoo, and F. Paraiso, "OCCIware - A Formal and Tooled Framework for Managing Everything as a Service," in *Projects Showcase @ STAF'15*, ser. Proceedings of the Projects Showcase @ STAF'15, CEUR, Ed., vol. 1400, L'Aquila, Italy, Jul. 2015, pp. 18 – 25.
[17] O.-W. A. Edmonds, "Open cloud computing interface-infrastructure," *Deliverable GFD*, vol. 184, no. 06, p. 270, 2011.
[18] S. Taherizadeh and V. Stankovski, "Dynamic Multi-level Auto-scaling Rules for Containerized Applications," *The Computer Journal*, p. bxy043, 2018.
[19] "Cron Expressions," https://docs.oracle.com/cd/E12058_01/doc/doc.1014/e12030/cron_expressions.htm.
[20] B. Zhang, Y. A. Dhuraibi, R. Rouvoy, F. Paraiso, and L. Seinturier, "CloudGC: Recycling Idle Virtual Machines in the Cloud," in *2017 IEEE International Conference on Cloud Engineering (IC2E)*, April 2017, pp. 105–115.
[21] S. D. Lowe. (2013) Best Practices for Oversubscription. [Online]. Available: https://communities.vmware.com/docs/DOC-34283
[22] "Flask," http://flask.pocoo.org.
[23] "Redis," https://redis.io.
[24] "Zabbix: Enterprise-class Monitoring Platform," https://www.zabbix.com, [Online; Accessed: 2017-06-02].
[25] W. TARREAU. (2013) HAProxy-The Reliable, High Performance TCP/HTTP Load Balancer. [Online]. Available: http://www.haproxy.org
[26] Synergy Research Group, Website https://www.srgresearch.com/articles/cloud-revenues-continue-grow-50-top-four-providers-tighten-grip-market, July 27, 2018.
[27] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, "Adaptive, Model-driven Autoscaling for Cloud Applications," in *11th International Conference on Autonomic Computing (ICAC 14)*. Philadelphia, PA: USENIX Association, 2014, pp. 57–64.
[28] S. Frey and W. Hasselbring, "The CloudMIG Approach: Model-Based Migration of Software Systems to Cloud-Optimized Applications," 2012.
[29] A. Ali-Eldin, J. Tordsson, and E. Elmroth, "An adaptive hybrid elasticity controller for cloud infrastructures," in *2012 IEEE Net. Operations and Management Symposium*, April 2012, pp. 204–212.
[30] M. Mohamed, M. Amziani, D. Belaïd, S. Tata, and T. Melliti, "An autonomic approach to manage elasticity of business processes in the Cloud," *Future Generation Comp. Syst.*, vol. 50, pp. 49–61, 2015.
[31] G. Copil, D. Moldovan, H. L. Truong, and S. Dustdar, "rSYBL: A Framework for Specifying and Controlling Cloud Services Elasticity," *ACM Trans. Internet Techn.*, vol. 16, pp. 18:1–18:20, 2016.
[32] K. Kritikos, J. Domaschka, and A. Rossini, "SRL: A Scalability Rule Language for Multi-cloud Environments," in *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, Dec 2014, pp. 1–9.
[33] A. B. Jrad, S. Bhiri, and S. Tata, "STRATModel: Elasticity Model Description Language for Evaluating Elasticity Strategies for Business Processes," in *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, 2017, pp. 448–466.

Yahya Al-dhuraibi received his Ph.D degree in computer science from the University of Lille, France in 2018. He has received his bachelor of science degree in computer science and engineering in 2010 from Aden University, Yemen and his Master degree in network system architecture from University of Evry, France in 2014. His current research focuses on cloud computing elasticity and automation in Scalair company and Inria - University of Lille, France.

Faiez Zalila is postdoctoral researcher at Inria Lille - Nord Europe within the Spirals research team. He contributes on the definition and the development of OCCIware Studio 2.0. He received the Ph.D. degree in computer science from National Polytechnics Institute of Toulouse (INP Toulouse) in 2014, and the M.Sc. degree in computer science from the University of Paul Sabatier Toulouse 3 in 2010. His research interests include model driven software engineering (MDE), software language engineering (SLE), software validation & verification (V&V) and Cloud Computing. For more information, see https://sites.google.com/site/faiezzalila/

Nabil Bashir received his Ph.D degree in computer science from the University of Rennes, France. He also received a diploma of computer science engineering from Batna Univ., Algeria and M.S. degree in computer science from the University of Rennes, France. His research interests include optimization, algorithms for constrained path computation, architectures and inter-carrier service delivery with assured QoS, complex event processing within Alcatel-Lucent Bell Labs and Inria. Nowadays, he is head of R&D and new technologies department at Scalair cloud provider. Current research is around cloud elasticity and resources optimization.

Philippe Merle is senior researcher at Inria and is member of the Spirals research team. He was associate professor at University of Lille 1, France. He obtained a PhD degree in computer science from University of Lille 1. His research is about software engineering for distributed systems, especially cloud computing, service oriented computing, middleware, model driven engineering, and component-based software engineering. He has co-authored two patents, two OMG specifications, one book, 15 journal papers, and more than 70 international conference papers.