A Model-Driven Framework for Interoperable Cloud Resources Management

Denis Weerasiri[1(✉)], Moshe Chai Barukh[1], Boualem Benatallah[1], and Jian Cao[2]

[1] University of New South Wales, Sydney, Australia
{denisw,mosheb,boualem}@cse.unsw.edu.au
[2] Shanghai Jiaotong University, Shanghai, China
cao-jian@cs.sjtu.edu.cn

Abstract. The proliferation of cloud computing has enabled powerful virtualization capabilities and outsourcing strategies. Suitably, a vast variety of cloud resource configuration and management tools have emerged to meet this needs, whereby DevOps are empowered to design end-to-end and automated cloud management tasks that span across a selection of best-of-breed tools. However, inherent heterogeneities among resource description models and management capabilities of such tools pose fundamental limitations when managing complex and dynamic cloud resources. In this paper we thus propose the notion of "Domain-specific Models" – a higher-level model-driven approach for describing elementary and federated cloud resources as reusable knowledge artifacts over existing tools. We also propose a pluggable architecture to translate these artifacts into lower-level resource descriptions and management rules. This paper describes concepts, techniques and a prototypical implementation. Experiments on real-world federated cloud resources display significant improvements in productivity. As well as notably enhanced usability achieved by our approach in comparison to traditional techniques.

Keywords: DevOps · Cloud resource management · Interoperability

1 Introduction

Cloud computing is evolving in both public and private cloud networks [14]. A third option involves a *hybrid* or *federated* cloud [2,16], drawing resources from both public and/or private clouds. The benefits include virtualization capabilities and outsourcing strategies. It is estimated that by 2016 the growth in cloud computing will consume the bulk of IT spend, whereby nearly half of all large enterprises will comprise hybrid cloud service deployments by end of 2017 [6].

However, exploiting cloud services poses great complexity. As development becomes increasingly distributed across multiple heterogeneous and evolving networks, it proves increasingly difficult to manage interoperable and portable

Q.Z. Sheng et al. (Eds.): ICSOC 2016, LNCS 9936, pp. 186–201, 2016.
DOI: 10.1007/978-3-319-46295-0_12

cloud resource solutions. Moreover, cloud applications inherently possess varying resource requirements during different phases of their life-cycle [9,14]. Consequently, designing effective cloud management solutions that cope with both heterogeneous and dynamic environments remains a deeply challenging problem.

Existing cloud management solutions typically rely on procedural programming (general-purpose or low-level scripting) languages [9,10,13,14,20]. Prominent examples include: *Puppet*, *Juju*, *Docker* and *Amazon OpsWorks*, [5]. This implies even DevOps (i.e. software and/or system engineers who design, develop, deploy and manage cloud applications) are forced to understand the different low-level cloud service APIs, command line syntax, Web interfaces, and procedural programming constructs - in order to create and maintain complex cloud configurations. Moreover, the problem intensifies with the increasing variety of cloud services, together with different resource requirements and constraints for each application. This inevitably leads to an inflexible and costly environment which adds considerable complexity, demands extensive programming effort, requires multiple and continuous patches, and perpetuates closed cloud solutions.

Drawing analogies from service representation (e.g. Web Service Description Language (WSDL)), and composition techniques (e.g. Business Process Execution Language (BPEL)), we are inspired to likewise support the abstract *representation* and *orchestration* of cloud resource by devising rich abstractions to reason about cloud resource requirements and their constraints. In this paper we therefore investigate how to effectively represent, organize and manipulate otherwise low-level, complex, cross-layer cloud resource descriptions into meaningful and higher-level segments. We believe this would greatly simplify the representation, manipulation as well as reuse of heterogeneous cloud resources. To enable this, we propose a methodology to support the automated translation of high-level resource requirements to underlying provider-specific resource and service calls. More specifically, this paper makes the following main contributions:

Domain-Specific Models to effectively *represent, manage* and *share* Cloud Resources. The ability to share and reuse cloud artifacts offer a powerful enhancement to DevOps' productivity. However, as these artifacts are inherently low-level and heterogenous between different cloud platforms, sharing such artifacts are almost useless in practice. To address this, we propose *Domain-specific Model*s (DSMs) for representing cloud resources and their management strategies as high-level entities. Based on the Entity-Relationship (ER) model, our proposed model features: a vocabulary and set of constructs for describing or representing both elementary (e.g. VMs, DBs, load balancers), and federated cloud resources (e.g. packaged virtual appliances); and their relationships (e.g. dependencies, configuration parameters, resource constraints). We architect this layer over existing cloud management platforms to harness interoperability capabilities. This means cloud resources could be easily combined to create higher-level virtual entities, called *Federated cloud resources*, thereby masking the complexity and heterogeneity from the underlying cloud services. For instance, by identifying common concepts among different tools, we can seamlessly merge those

features for end-to-end configuration. For example, a VM deployed by *Vagrant* can be modified by another tool, such as *Puppet* with fine-grained configuration tasks (e.g., installing software within the VM) that are not supported by the initial tool. We assume one particular DevOps who is an expert of a particular cloud tool would specify the associated *Domain-specific Model* at the onset.

Connectors for *automated translation* of DSM-based models into native resource artifacts. Connectors accomplish the magic behind the scenes. As mentioned, there are a large variety of resource representation *languages* (e.g. procedural, activity based and declarative); as well as several different types of *tools/APIs* to manage/orchestrate these resources; that all may need to adapt to different *environments* (i.e., public, private and federated). This three-fold level of heterogeneity make cloud resource management a tedious task. We therefore propose the notion of *Connectors*, which provide a high-level interface (i.e. API) for DevOps to deploy, configure and manage cloud resources. As mentioned, the proposed Domain-Specific Model could be used to represent resource configuration using high-level entities and relationships. Behind the scenes, connectors are thereby able to: (a) translate these high-level descriptions into their native format (e.g., files, shell code snippets); and (b) interpret what are the required management operations and transform them into low-level API calls. For example, to create an Image using *Docker's Remote API*, traditionally DevOps would need to be skilled in the tool's communication protocol[1]. This is alleviated using our proposed approach. In addition, *Connectors* may include basic events that are to be monitored by periodically querying for data using low-level APIs. DevOps are thus empowered to write automated management processes such as Event-Condition-Action (ECA) rules and workflows over the operations exposed by the *Connectors*. We assume *Connectors* are implemented by DevOps, who have expertise in programming and knowledge on the particular cloud tool.

The rest of this paper is organized as follows: In Sect. 2, we investigate in the context of example scenarios, specific limitations amongst existing cloud management techniques. In Sect. 3, we present our proposed system architecture. In Sect. 4, we elucidate our domain-specific model, with a case-study on *Docker*. In Sect. 5, we discuss our implementation, while in Sect. 6 we present our implementation and evaluation. In Sect. 7, we examine related work, and conclude with remarks and a discussion of future work.

2 Limitations in Current Cloud Management Solutions

As mentioned, current cloud management solutions rely on low-level script-based languages. For example, Ubuntu Juju employs *Charms*[2]; and similarly Docker employs *Dockerfiles*[3]). Charms and Dockerfiles are a collection of configuration attributes and executable scripts that configure, install and start an application. Inevitably constructs of these scripts typically include basic commands

[1] http://docs.docker.com/reference/api/docker_remote_api/.
[2] https://jujucharms.com/.
[3] https://docs.docker.com/reference/builder/.

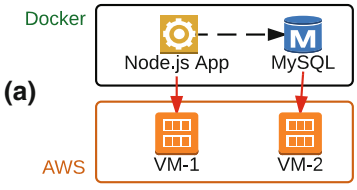
(e.g., `RUN`, `COPY`, `CMD`), which provide little or no abstraction for DevOps to identify the main attributes and relationships of the constituent cloud resources.

Scenario 1. Consider describing composite cloud resources: A Web Application stack, with a *Node.js* application engine and *MySQL* database. DevOps could describe attributes (e.g. memory, size) of these resources, as well as their relationship (e.g. app engine stores data in database), using *Dockerfiles*. Docker provides either a RESTful or CLI interface to interpret *Dockerfiles* in order to build, deploy, monitor and control necessary resources known as *Containers* on a given Virtual Machine (VM) (refer to Fig. 1(a)). However, as Docker does not support configuring and deploying VMs[4], another cloud management tool would be needed, such as *AWS-EC2 CLI* or *Rackspace CLI*. This therefore forces DevOps to employ multiple tools to automate end-to-end management tasks.

Moreover, considering that every cloud management tool employs their own resource description models, management capabilities and interfaces – the challenges described above only increases several-fold. For example, Fig. 1(b) lists the variety of heterogeneous configuration and management interfaces exposed by different tools. Consequently, these ad-hoc scripts introduce hard-coded dependencies among resources that are orchestrated by different tools. Reusing knowledge artifacts, which include such ad-hoc scripts is not scalable as DevOps are required to manually analyze those knowledge artifacts in order to apply cross-domain relationships among resources within a composite cloud solution.

Scenario 2. This time consider the case of federated cloud resource management. For example, VMs deployed and managed amongst two different cloud services, such as AWS and Backspace. If additional VMs would want to be added (in order to improve reliability and handle increasing demands of the Web application), DevOps would need to implement additional orchestration scripts that monitor the application load and deploy the Web application in either AWS or Rackspace based on a certain load-balancing algorithm. However, both AWS and Rackspace employ different formats of access credentials and management interfaces to deploy VMs.

Overall the level of heterogeneity amongst current cloud solutions entails great complexity when exploiting cloud services. More specifically, with existing cloud delivery models, developing a new cloud-based solution generally leads



(b)

Tool	Type of Interface Available
AWS	CLI, SDKs, Web 2.0, REST and SOAP APIs
Rackspace	CLI, SDKs, Web 2.0, REST API
Puppet	CLI, Web 2.0, REST API
Docker	CLI, REST API

Fig. 1. (a) Components and Relationships of a *Node.js* Web application Stack; (b) List of available cloud Configuration and Management Interfaces

[4] This feature was only later introduced by Docker (the principle remains the same).

to uncontrollable fragmentation across the use of different cloud languages and tools (e.g., Puppet, Chef, Juju, Docker, SmartFrog, AWS OpsWorks) [4,5,7,8]. This makes it very difficult to develop interoperable and portable cloud solutions. It also degrades performance as applications cannot be partitioned or migrated easily into another cloud platform when demand cycles increase.

3 Next-Generation Cloud Resource Management: Architecture Overview

The next-generation in cloud resource management with require ease of interoperability - enhanced productivity with a viable opportunity for reuse. The limitations mentioned above is an immense setback. To overcome this, we propose a layered architecture (see Fig. 2) that enables: (a) *Domain-Specific Models (DSM)* (for high-level representation and management models of cloud resources; and (b) *Connectors* (to automate translation of these high-level DSMs into low level resource descriptions and management scrips.

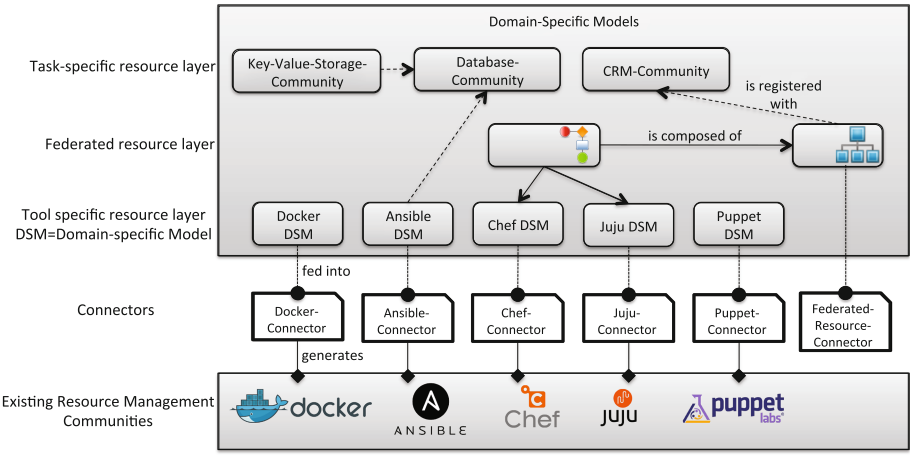


Fig. 2. System Overview

Domain-Specific Models layer consists of three sub-layers: (i) Tool-specific resource layer; (ii) Federated resource layer; and (iii) Task-specific resource layer. All sub-layers consist of a collection of DSMs. Starting from bottom-up, the *tool-specific resource layer* include DSMs that represents cloud resource entities (e.g., resource descriptions, management rules) and relationships among those entities of a 'particular' cloud tool. For example, *Docker DSM* describe linked entities that are provided specifically by the Docker engine. Tool-specific *Domain-specific Models* can also be combined to create higher-level DSMs that represent *federated cloud resources*, which may be managed by two or more existing cloud tools.

For example, a customer relationship management application of an organization, which is deployed in a public cloud service (e.g., AWS), may access a client information database server, which is managed within the organization's private cloud infrastructure (e.g., VMWare). Finally, the *task-specific resource layer* represent "splices" of the fundamental DSM that are reformulated to specific types of categories. For example, DSMs for the *Database Community* may include models that facilitate key-value storages, relational databases and graph databases. The extended goals of DSMs are also to abstract unwanted heterogeneous notations in order to simplify for the end-developer. DSMs can thus be customized to further accommodate this.

Connectors layer, are essentially the glue between the high-level DSM model and underlying cloud tool – it serves to abstract an otherwise complex and heterogenous interface into a simplistic and interoperable one. Connectors assume a DSM has been defined for a particular cloud tool. As we will describe in the next section, a particular DSM has both a *description model* and actionable *management model*. Connectors utilize both these models to auto translate high-level calls into low-level actions. We have observed every cloud tool supports three basic operations: *deploy*, *control* (or *reconfigure*) and *undeploy*. Accordingly, we have built Connectors to function with the following interface:

`init(resource_model)`: Translates a high-level resource DSM-based "description model" into its native script, (e.g. files, shell code) and returns a `unique-ID`.

`deploy(resource_model)`: The runtime selects a particular connector implementation that can deploy the inputted resource configuration model. The connector implements runs the tool-specific `deploy` command.

`control(resource_ID, actions)`: Actions are also described using the high-level DSM-based "management model". The connector implementation maps this into low-level API calls to apply over the inputted `resource_ID`.

`undeploy(resource_ID)`: The runtime detects the appropriate Connector and calls the `undeploy` operation over the specified resource.

Additionally, connectors are also vital for enabling dynamic control. As we will describe in the next section, our "management model" also support events (e.g., connection failure to VM), and DevOps may annotate resource configurations with simple rules that trigger actions upon particular events.

4 Extracting Domain-Specific Models

As mentioned, we have adopted the *Entity-Relationship (ER)* notation to represent *Domain-Specific Models*. The process of building DSMs for a particular tool involves analyzing existing knowledge sources (e.g. language specifications, user documentations, forums and resource description repositories) to understand key entities for describing resources. We assume at least one domain expert would contribute this for a particular, which may then be reused multiple times by other DevOps. Next, we extracted relationships between the entities by understanding how entities are associated when describing composite cloud resources.

Similarly, we extracted what *actions* and *events* are provided by these tools, such as for manipulating the given resource. These events and actions allow DevOps to annotate resource descriptions with ECA rules.
In essence, our embryonic data-model consists of the following elements:

1. **Resource Description Model:** It describes cloud resources in terms of relevant entities (and their attributes), as well as their interconnection of relationships. For example, a VM entity may include CPU, memory and storage as attributes.
2. **Resource Management Model:** It allows to specify cloud management operations, particularly to configure, deploy, monitor and control cloud resources. This model consists of two sub-models:
   (a) ***Action Model:*** It specifies available actions (e.g., deploy, configure, migrate) to manage cloud resources. It is expressed a set of entities with relevant attributes that express required input and output parameters.
   (b) ***Event Model:*** It expresses events related to the lifecycle of cloud resources in terms of entities with necessary attributes that describe events [14]. It should be noted that, the issues of event detection while important, are complementary to the research issues addressed in our work and thus outside the scope of this paper.

The benefit of the model, as stated earlier, is that DSMs enables DevOps to work with a high-level design that captures cloud resources as entities and relationships. Concrete cloud configurations can be described based on the DSM. Additionally, DSMs provide a lightweight documentation approach. In contrast with existing script-based approaches, complex resource configurations are often only documented separately in form of ad-hoc Wikis that quickly gets outdates unless continuously maintained. Additionally, our ER-based model inherently supports machine-readable syntax, which can be consumed by software like Connectors to automatically generate cloud resource descriptions, deployment and management scripts.

4.1 *Docker* Case-Study

We built DSMs for a diverse range of tools and languages, including: *Docker*, *Juju* and *TOSCA*. Due to space constraints, we have chosen Docker as a case-study to illustrate in this paper (see Fig. 3). *Docker* is an open-source and emerging industry standard. (Example of other models have been published online[5]

By analyzing the provided specifications[6,7], we identified six key resource description entity types: (1) `Container`, (2) `Image`, (3) `Application`, (4) `Registry`, (5) `Hosting-Machine` and (6) `Cluster` (refer to Fig. 3).

[5] http://mosheb.web.cse.unsw.edu.au/DSM/appendix.html (Appendix of this paper has been published online for readers' further interest).
[6] https://docs.docker.com/reference/builder/.
[7] https://docs.docker.com/compose/reference/.

The central entity: `Container` represents a virtualized software container where DevOps may deploy an application. Deployment knowledge of the application and its dependencies is represented via the entity, `Image`. Such knowledge is either represented using one monolithic `Image` instance or a set of `Image` instances. In other words, the `Image` possesses deployment knowledge required to instantiate a `Container`. An `Application` represents a logical entity that includes a collection of related `Containers`. Each `Container` constitutes a component of the `Application`. The entity `Registry` represents a repository of `Image`s where DevOps organize, curate and share resource deployment knowledge. The entity `Hosting-Machine` represents the location where a Container is hosted (e.g., VM or physical machine). A `Cluster` represents a set of `Hosting Machines`. This reduces the overhead of dynamically managing multiple machines. For example, the `Cluster` may automatically decide which `Hosting Machine` will be chosen to deploy the given container based on an optimization algorithm [15].

We then derive the *attributes* that characterize each entity; and the *relationships* amongst them. For example, the relationship between `Hosting-Machine` and `Container` is `Deployment`. The `Containment` relationship defines the hierarchical organization of entities. For example, a `Containment` relationships exist between a `Container` and its related `Application`; and between a

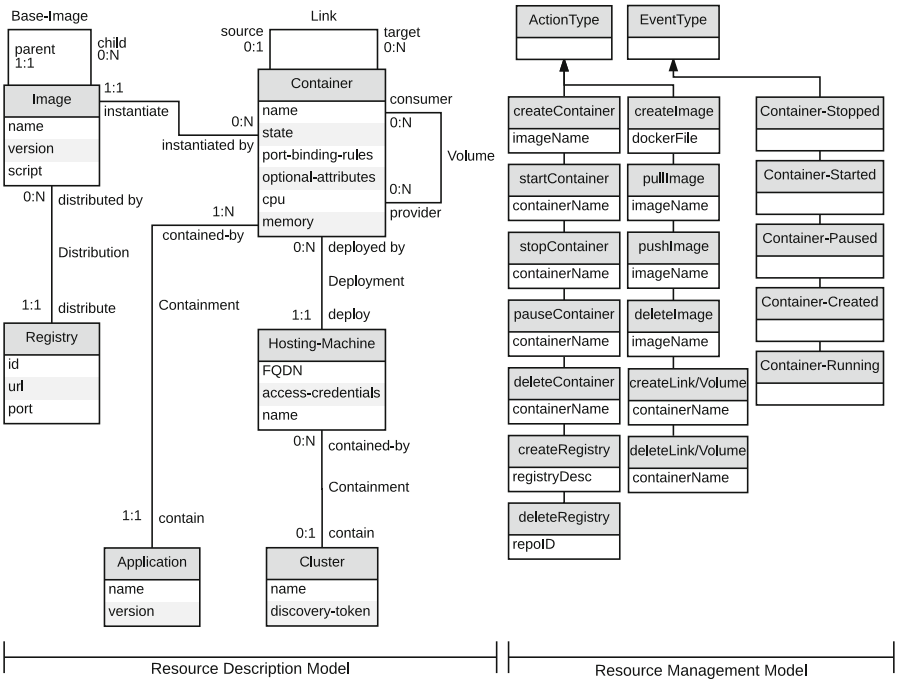


Fig. 3. *Domain-specific Model* for Docker

`Hosting-Machine` and its related `Cluster`. Similarly, we derive all other relevant relationships.

Next we extract *actions*. For example, `create`, `start`, `stop`, `pause` and `delete` to manipulate `Containers`. As well as all other actions that Docker offers.

We then extract basic *events* that are supported[8]. Such as: `@Created`, `@Started`, `@Stopped`, `@Paused`, `@Running` and `@Killed`, to detect the runtime state of `Containers`.

We then specify additional events that are not directly supported by the tool, but are required by the *Connector* for resource management. For example, we may specify a periodic event that includes memory usage data of a particular `Container`. In addition, we may specify composite events based on previously extracted events using an existing event-pattern specification language (e.g., Esper EPL). For example, we may specify a composite event, which gets triggered if the memory usage of a `Container` exceeds 95 % and then the `Container` is killed, to identify `Containers` that get crashed due to the shortage of available memory.

5 Implementation

Curating DSMs and Connectors. Our design is based on crowd-driven incremental contributions, whereby domain experts of a particular tool collectively participate in curating (i.e. creating and/or updating) DSMs. In our current implementation, we serialize *entities* and *relationships* using *JSON-Schema*. DSMs therefore produce high-level cloud resource schemas that enforce constraints over entity attributes (e.g. datatype, optionality); and relationships (e.g. cardinality). Additionally, actions and events of DSMs may also be defined. Complex or customizable events are also supported using *Esper EPL*[9].

Similarly, experts may also contribute *Connectors*, by providing the necessary business logic for each specified operations that make up the connector interface. Earlier at Sect. 3, we defined a generic programmable interface for DevOps to implement connectors, this include four mandatory operations. Additionally, a Connector may have any number of operations that implement the actions specified in the relevant DSM. For example, Docker defines a `createContainer` method, which: (i) accepts the name of an `Image`; (ii) prepare the `Hosting-Machine` to deploy a `Container`; and (iii) invoke `docker run` command in the Docker CLI[10] along with an `Image`. A snippet of a connector interface is shown at Fig. 4.

Once both the *DSM* and *Connector/s* are registered, DevOps are then able to create cloud resources and moreover, implement management processes/rules based on the simplicity offered by the high-level DSM.

User-Interface. Our current implementation provides a *Command-Line Interface (CLI)*, as well as a prototypical *Graphical User Interface (GUI)*. Our GUI

[8] https://docs.docker.com/engine/reference/commandline/events/.
[9] http://rsper.codehaus.org.
[10] https://docs.docker.com/reference/commandline/cli/.

includes *DSM Editor* that enables curators to *graphically* specify the structure of entities, relationships, actions and events when describing a DSM. We reuse Java-script library named *JSON Schema Based Editor*[11] for the generation and verification of JSON schemas.

The CLI can also accomplish DSM specifications albeit programmatically. In addition, the CLI enables DevOp to invoke operations. For example, calling the `init` and `deployContainer` actions in order to deploy a Container within the Docker runtime (refer to Code 1.1). Behind the scenes an appropriate connector is selected, which fires the action call. DevOps are required to first locate the DSM, as shown at Listing 1.1.

Code 1.1. CLI command to deploy a container in Docker

```
cd ~/base-git-repo/node-app-1  #location of the JSON-based
    resource description
cloudbase docker.rest -action=init
cloudbase docker.rest -action=deployContainer
                      -input={"resource":"node-engine.json"}
```

Automated Translation into Native Artifacts. Figure 4 provides an illustrative overview. The *translation logic* utilizes both the preregistered DSM and Connector/s. A DevOp may then describe a resource configuration: in this example, a `MySQL` and `Node.js` Web application (based on *Scenario 1* we described in Sect. 2). Subsequently, a `Dockerfile` and `build.sh` file is automatically generated for each resource. Note: The files shown in Fig. 4 are for illustrative purposes only. For larger print, readers are directed to our online appendix (See footnote 5).

Similarly, the *build.sh* file is generated based on a sequence of commands, which: (a) reads the *Dockerfile*; (b) generates a concrete Image; (c) uploads the generated Image to a specified `Registry` (i.e., *Registry-1*); and (d) creates a concrete Container from the concrete Image in a specified `Hosting-Machine` (i.e., *HostingMachine-1)*. The *build.sh* file may also include commands to instantiate relationships (e.g. `Links` and `Volumes`) between dependent concrete Containers. The transformation logic extracts the required input data for these commands from attributes that were defined in Docker's DSM.

In addition, management scripts such as event/action rules are also auto-translated into low-level API calls. For example, a management rule, that creates new `Containers` to handle increasing load is depicted at the bottom of Fig. 4. The generated file depicts a sequence of API calls which: (a) logs-in to a particular `Hosting-Machine`; (b) creates an `Image` (if it doesn't already exist); and (c) creates and starts a `Container`.

Storage. We utilize a *JSON Object Store*; a Git[12] repository to store and share *DSMs* and their objects as JSON files. Related cloud resource descriptions are organized into separate folders within the repository. The *JSON Object Store*

[11] https://github.com/jdorn/json-editor.
[12] https://git-scm.com/.

allows keeping multiple versions of JSON files and trigger events when certain modifications (e.g., store, update, delete cloud resource descriptions) occur. These features are very useful to dynamically reconfigure cloud resources and roll-back to a previous stable configuration if an error occurs.

Event Management System. We detect and process lower-level monitoring events (e.g., (re)starting, CPU and memory usage) from different cloud services (e.g., Docker, AWS), and thereby generate higher-level events for DevOps. Events to be collected are defined as part of the DSM. We support both *Pull* and

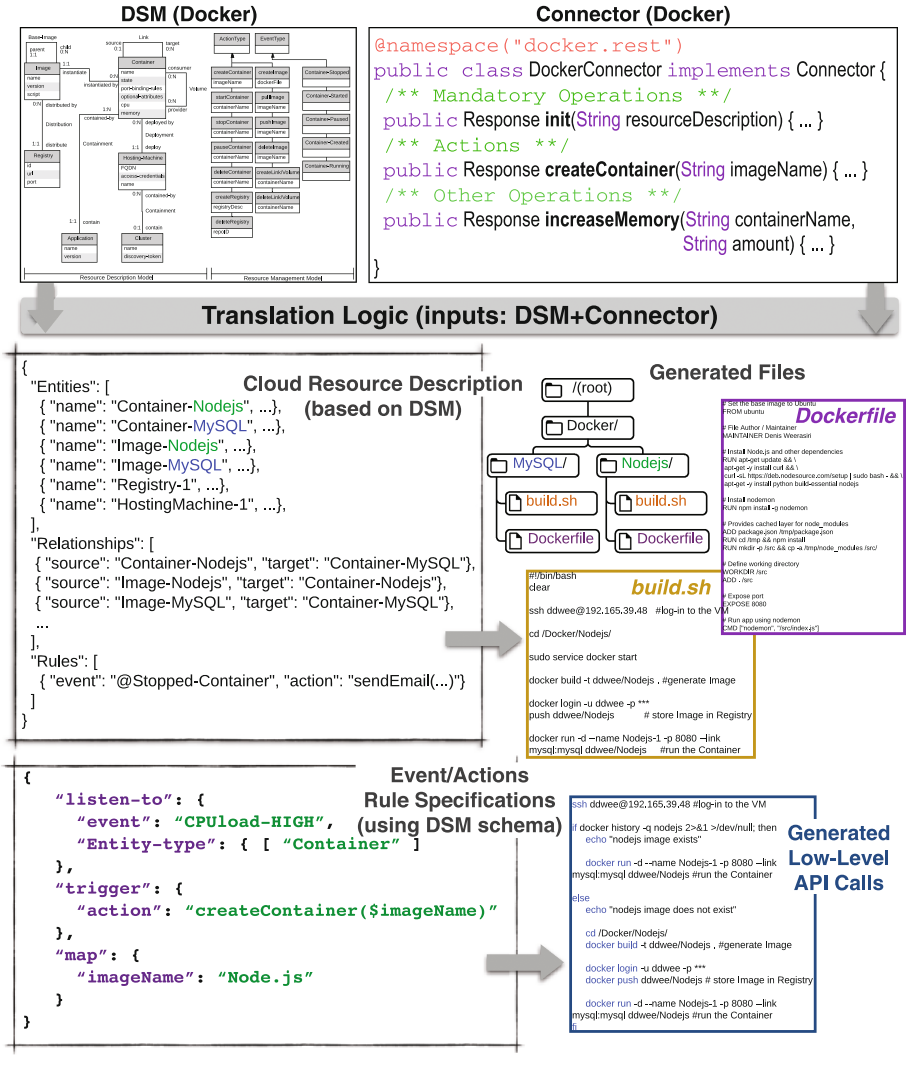


Fig. 4. Automated translation of high-level models into native artifacts

PuSH -based detection. We use *Fluentd*[13] to perform *polling* and extract JSON-based events. For example, events related to state changes of `Containers` as per Docker's DSM. For extracting *PuSH*-based events, we leverage Apache Camel[14]. For specifying, processing and generating high-level events we use *Esper EPL*. High-level events enable defining events based on a series of low-level events. For example, *Esper* may trigger an event named `CPULoad-High` for a particular Application in Docker if the CPU usage of each Container of the Application is over 95 %, (as illustrated in Fig. 4). We also implement a Java-based event publishing channel for consumers (such as the *Rule Processor*) to subscribe.

Rule Processor. We also support automation capabilities via simple reactive rules. For example, DevOps may specify *if* `@Stopped` *then* `#notify`, which implies if some resource has stopped, perform some notification action. We greatly simplify the definition of by reusing our previous work [3], where we adopted a *"knowledge-based"* approach, which means APIs and their constituents (i.e. operations, input/output types) of the orchestration tools are loaded in a knowledge-base. This makes it possible to write high-level rule definitions and translate into concrete actions. At Fig. 4, we showed a simple rule. The `listen-to` construct specifies events to detect; the `trigger` construct specifies what *actions* to invoke; and the `map` construct describes the required input parameters for the invocation.

6 Evaluation

We conducted a user-study to evaluate the following hypotheses: ***H***, the *Domain-Specific Model* approach is more *efficient* to *accurately* configure and deploy cloud resources. We measured efficiency as the time taken to complete the tasks and the number of lines-of-code excluding whitespace; whereas accuracy was determined by deploying each cloud resource description and checking whether the resultant deployment complied with the initial deployment specification.

Participant Selection and Grouping. Participants were sourced with diverse levels of technical expertise. For the sake of analysis, we classified a total of 14 participants into 2 main groups: (I) Experts (8 participants) with sophisticated understanding of cloud orchestration tools with 2–8 years of experience. And (II) Generalists (6 participants) who have average knowledge of cloud orchestration tool for day-to-day requirements, with around 1–5 years of experience.

Use-Case. We asked participants to configure and deploy the following scenario: A platform that requires an AWS-EC2 VM where a Docker Container resides within. The container includes Redmine[15], a project management service, and a Git client[16]. The Redmine service is intended to: (i) extract commits from a

[13] http://www.fluentd.org/.
[14] http://camel.apache.org/.
[15] http://www.redmine.org/.
[16] http://git-scm.com/.

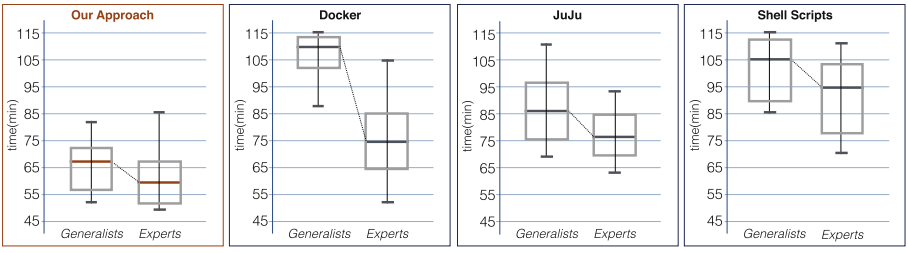
specified source repository in GitHub via the Git client; and (ii) link them with relevant bug reports. In addition an AWS-S3 bucket (i.e. key-value store), which acts as a software distribution repository, is required.

Experimental Setup. Prior to the experiment, participants attended an individual training session, where our tool was explained via a hands-on presentation. We also explained them the use-case scenario. For quantitative comparison purposes, we conducted the same experiment against two third-party tools: *Docker* and *Juju*. Only a total of 8 and 5 out of 14 DevOps participated in the *Docker* and *Juju* based experiments respectively. This was due to some DevOps not having expertise and confidence to use those tools. In addition, a total of 7 participants implemented the same deployment specification using Shell scripts to estimate an upper bound of the test results.

6.1 Experiment Results and Analysis

Evaluation of H. The hypothesis $\boldsymbol{H}$ was evaluated based on the time taken and number of lines-of-code. Alternatively, we sought to disprove the null hypothesis $\boldsymbol{H}_0$. The hypothesis was examined by conducting a t-test with a probability threshold of 5 %, and assuming unequal variance.

As shown in Fig. 5, it was pleasantly surprising that even generalists demonstrated a significant increase in efficiency (i.e. reduction in time and lines-of-code). More specifically, the time taken to complete the task was reduced by 31 % in comparison to there other approaches. Similarly, the number of lines-of-code was reduced by 37.2 %. Participants reported that they much rather preferred an *entity-relationship (ER)* based abstraction for describing resources, as opposed to script-based languages that are provided by otherwise widely adopted cloud management tools, such as *Docker* and *JuJu*. DevOps confirmed this greatly helped improve their configuration and deployment time.



	Hypothesis (H): All Participants			
	Our Approach	Docker	JuJu	Shell Scripts
Mean	63	93	80	101
Variation	87.69	69.42	203.5	175.77
Observations	14	8	5	7

	Hypothesis (H)
df	18
p (T ≠ t)	0.0003
t Critical two-tail	26
Reject H_0	YES

	#LOC
Our Approach	82
Docker	126
JuJu	113
Shell Scripts	153

Fig. 5. Results (Time, grouped by expertise); *t-test* Results; and *Lines-of-Code*

On the other hand, our approach assumes that appropriate *Domain-Specific Models* and *Connectors* have been defined and registered. This does incur additional costs to implement, however we argue this is typically a one-off for the benefit of many. Once registered, countless DevOps would benefit over many occasions. Moreover, our *knowledge-driven* approach implies knowledge (such as high-level representations of cloud resource configurations) can be incrementally shared and collectively reused, which significantly improve productivity to implement federated management spanning across multiple cloud services.

Due to the vast number of alternative tools, and project-based constraints, a more exhaustive comparative experiment was outside the scope. However, given the notable differences in times (mean of 63, against 93, 72 and 101 min), we postulate it is unlikely to observe fundamental differences when comparing with any other tools similar to Docker or Juju. Accordingly, given our observations the likelihood of $\boldsymbol{H}_0$ (equal mean modeling time) was around 5 %. Therefore, we could safely reject these null hypotheses, and imply the truth of $\boldsymbol{H}$.

7 Related Work and Concluding Remarks

Tools such as, *Puppet*, *Chef*, *Juju*, *Docker*, *SmartFrog* and *AWS OpsWorks*, as well as various research initiatives [4,5,7,8,19], all provide domain specific languages to represent and manage resources in a cloud environment. These languages are either template-based or model-driven [11].

Cloud Resource Representation and Management Languages. Template-based approaches (e.g., Open Virtualization Format) aggregate resources from a lower-level of the cloud stack and expose the package, along with some configurability options, to a higher-layer. Model-driven approaches (e.g., TOSCA [12]) define various models of the application at different levels of the cloud stack, and aim to automate the deployment of abstract pre-defined composite solutions on cloud infrastructure [9,14]. Our approach proposes *Domain-specific Model*s – a methodology to extract cloud resource management entities from such model-driven and template-based languages. In contrast, our approach invites an interoperable vocabulary to build elementary and federated cloud resources, as an *abstract*-layer over these "multiple and diverse languages".

Enabling Federated Cloud Management. Federation of cloud resources implies building cross-provider solutions. For example, `Hosting-Machine` in *Docker* represents a VM where `Containers` are deployed, albeit the *Docker* run-time itself cannot provision VMs. *JuJu* on the other hand focuses on managing a set of VMs, and can provision these VMs. To support this, we require a middleware that either: (i) defines a unified cloud resource language; or (ii) provide a pluggable architecture that accepts and interprets different resource models. The former is clearly not feasible, would be costly and require existing tools to undergo major architectural changes or complex model transformations to conform to a new language provided by the middleware. We thus believe the latter approach provides a more pragmatic and adaptive solution that can be

integrated amongst a set of already existing and prevalent tools. To solve this gap, we thus precisely propose the notion of Domain-specific models, to automate end-to-end deployment (e.g. Docker Containers on VMs which are provisioned by Juju).

Model-Driven Approach and Combating Heterogeneity. *TOSCA* is an open-standard for unified representation and orchestration of cloud resources [12]. Wettinger et al. proposes a model transformation technique that generates TOSCA-based descriptions from resource descriptions in *Chef* and *Juju* [18]. *MODAClouds* [1] is another approach to design and manage multi-cloud applications. It proposes four layers that incrementally transform functional and non-functional requirements of applications into tool-specific resource tasks. Konstantinous et al. [8] presents a description and deployment model that first models a resource as a provider-independent resource configuration, called "Virtual Solution Model", and then another party can transform the provider-independent model to a provider-specific model called, "Virtual Deployment Model". However, this approach only allows users to compose federated resource configurations from a single provider for a single deployment. In contrast, our approach considers the resource federation from multiple providers as a first class citizen.

Summary. The "cloud" plays an increasingly vital role in modern-computing technology. Accordingly, a vast variety of configuration and management tools have been proposed, albeit they differ with respect to representation language, as well as user-interface. Overall they assume a low-level and sophisticated programmatic approach. The paper provides an innovative approach for dealing with this: Firstly, in stead of competing with existing approaches, we embrace them by providing a "higher-level" and "interoperable" layer via the notion of Domain-specific Models. Such models can recompose to even higher-level models, in order to capture a particular use-case. Moreover, we encourage a knowledge-sharing paradigm unlike any other existing approach. We realized our approach via a pluggable architecture (i.e., *Connectors*) – a programmable interface that allows DevOps to deploy and manage high-level cloud resource representations. Behind the scenes, *Connectors* translate high-level models into native scripts. We evaluated our work with a user-study that yielded significantly promising results. We are therefore confident our work provides an innovative approach to a new way of cloud management. As future work, we plan to integrate a recommender system, and visual notations based on our previous work [17].

References

1. Ardagna, D., et al.: Modaclouds: a model-driven approach for the design and execution of applications on multiple clouds. In: MISE, pp. 50–56, June 2012
2. Bahga, A., Madisetti, V.K.: Rapid prototyping of multitier cloud-based services and systems. Computer **46**(11), 76–83 (2013)
3. Barukh, M.C., Benatallah, B.: *ProcessBase*: a hybrid process management platform. In: Franch, X., Ghose, A.K., Lewis, G.A., Bhiri, S. (eds.) ICSOC 2014. LNCS, vol. 8831, pp. 16–31. Springer, Heidelberg (2014)

4. Chieu, T.C., et al.: Solution-based deployment of complex application services on a cloud. In: SOLI, pp. 282–287. IEEE (2010)
5. Delaet, T., Joosen, W., Vanbrabant, B.: A survey of system configuration tools. In: 24th International Conference on LISA, pp. 1–8. USENIX Association (2010)
6. Gartner says cloud computing will become the bulk of new it spend by 2016. http://www.gartner.com/newsroom/id/2613015. Accessed 07 Dec 2014
7. Goldsack, P., et al.: The smartfrog configuration management framework. ACM SIGOPS Oper. Syst. Rev. **43**(1), 16–25 (2009)
8. Konstantinou, A.V., et al.: An architecture for virtual solution composition and deployment in infrastructure clouds. In: VTDC, pp. 9–18. ACM (2009)
9. Liu, C., Loo, B.T., Mao, Y.: Declarative automated cloud resource orchestration. In: Proceedings of the SOCC 2011, pp. 1–8. ACM (2011)
10. Lu, H., et al.: Pattern-based deployment service for next generation clouds. In: 2013 IEEE Ninth World Congress on SERVICES, pp. 464–471, June 2013
11. Misic, V., et al.: Guest editors' introduction: special issue on cloud computing. IEEE Trans. Parallel Distrib. Syst. **24**(6), 1062–1065 (2013)
12. OASIS: Topology and Orchestration Specification for Cloud Applications (TOSCA), Version 1.0 (2013)
13. Ponge, J., Benatallah, B., Casati, F., Toumani, F.: Analysis and applications of timed service protocols. ACM Softw. Eng. Methodol. **19**(4), 11:1–11:38 (2010)
14. Ranjan, R., Benatallah, B.: Programming cloud resource orchestration framework: operations and research challenges. CoRR abs/1204.2204 (2012)
15. Schulte, S., Janiesch, C., Venugopal, S., Weber, I., Hoenisch, P.: Elastic business process management: state of the art and open challenges for BPM in the cloud. Future Gener. Comput. Syst. **46**, 36–50 (2015)
16. Veeravalli, B., Parashar, M.: Guest editors' introduction: special issue on cloud of clouds. IEEE Trans. Comput. **63**(1), 1–2 (2014)
17. Weerasiri, D., Barukh, M.C., Benatallah, B., Jian, C.: *Cloudmap*: a visual notation for representing and managing cloud resources. In: Nurcan, S., Soffer, P., Bajec, M., Eder, J. (eds.) CAiSE 2016. LNCS, vol. 9694, pp. 427–443. Springer, Heidelberg (2016)
18. Wettinger, J., Breitenbucher, U., Leymann, F.: Standards-based devOps automation and integration using TOSCA. In: 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC), pp. 59–68, December 2014
19. Wilson, M.S.: Constructing and managing appliances for cloud deployments from repositories of reusable components. In: Proceedings of the 2009 Conference on HotCloud 2009. USENIX Association (2009)
20. Zeng, L., et al.: QoS-aware middleware for web services composition. IEEE Trans. Softw. Eng. **30**(5), 311–327 (2004)