A Taxonomy and Survey of Cloud Resource Orchestration Techniques

DENIS WEERASIRI, MOSHE CHAI BARUKH, and BOUALEM BENATALLAH,
University of New South Wales, Australia
QUAN Z. SHENG, Macquarie University, Australia
RAJIV RANJAN, University of Newcastle, United Kingdom

Cloud services and applications prove indispensable amid today's modern utility-based computing. The cloud has displayed a disruptive and growing impact on everyday computing tasks. However, facilitating the orchestration of cloud resources to build such cloud services and applications is yet to unleash its entire magnitude of power. Accordingly, it is paramount to devise a unified and comprehensive analysis framework to accelerate fundamental understanding of cloud resource orchestration in terms of concepts, paradigms, languages, models, and tools. This framework is essential to empower effective research, comprehension, comparison, and selection of cloud resource orchestration models, languages, platforms, and tools. This article provides such a comprehensive framework while analyzing the relevant state of the art in cloud resource orchestration from a novel and holistic viewpoint.

Categories and Subject Descriptors: H.3.5 [**Online Information Services**]: Web-based services

General Terms: Methods, Techniques, Tools

Additional Key Words and Phrases: Cloud computing, resource orchestration, Service oriented architectures

ACM Reference Format:
Denis Weerasiri, Moshe Chai Barukh, Boualem Benatallah, Quan Z. Sheng, and Rajiv Ranjan. 2017. A taxonomy and survey of cloud resource orchestration techniques. ACM Comput. Surv. 50, 2, Article 26 (May 2017), 41 pages.
DOI: http://dx.doi.org/10.1145/3054177

1. INTRODUCTION

As economies undergo significant structural change, organisations are competitively compelled to leverage cloud computing to expand or contract their computing footprint based on variable demands for computing resources [Wang et al. 2012; Cui et al. 2013; Bahga and Madisetti 2013]. Typically, cloud providers enable virtualising three categories of resources—the "cloud computing stack" [Armbrust and et al. 2010]—which includes the following: *Software* (e.g., user facing applications), *Platform* (e.g., development and runtime environments), and *Infrastructure* (e.g., storage, networking and hosting) [Ranjan et al. 2015; Satzger et al. 2013; Weerasiri and Benatallah 2015]. Accordingly, cloud computing is evolving in the form of both public (deployed by IT organisations) and private clouds (deployed behind an enterprise firewall). A third

We gratefully acknowledge funding for this research by the Australian Government through the Australian Research Council/Discovery Project (DP150102966, Federated Cloud Services Configuration and Orchestration).
Authors' addresses: D. Weerasiri, M. C. Barukh, and B. Benatallah, School of Computer Science and Engineering, University of New South Wales, Sydney NSW 2052, Australia; Q. Z. Sheng, Department of Computing, Macquarie University, Sydney, NSW 2109, Australia; R. Ranjan, School of Computing Science, Claremont Tower, Newcastle University, NE1 7RU, United Kingdom.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2017 ACM 0360-0300/2017/05-ART26 $15.00
DOI: http://dx.doi.org/10.1145/3054177

option, a hybrid or federated cloud [Bahga and Madisetti 2013], draws computing resources from one or more public clouds and one or more private clouds, combined at the behest of its users. A Gartner report [Gartner 2013] estimates "nearly half of all large enterprises having cloud service deployments by the end of 2017."

However, there are crucial gaps in the cloud-enabled endeavor [Ranjan et al. 2015; Satzger et al. 2013; Lu et al. 2013; Ranjan et al. 2013]. Modern orchestration frameworks like *Puppet*, *Ubuntu Juju*, *Ansible*, *Amazon OpsWorks*, and *Chef* provide scripting-based languages for describing cloud configurations [Delaet et al. 2010]. Consequently, even sophisticated programmers are forced to understand various low-level cloud service Application Programming Interfaces (APIs), command lines, and procedural programming constructs to create and maintain complex resource configurations. This leads to a costly environment that lacks flexibility and is significantly more complex. It implies extensive programming effort, requires multiple and ongoing patches, and perpetuates closed-cloud solutions. This intensifies as the variety of cloud services and resource requirements increase. More specifically, with existing cloud delivery models, developing a new cloud-based solution often leads to uncontrollable fragmentation. This makes it very difficult to develop interoperable and portable cloud solutions. It also degrades performance as applications or workloads cannot be partitioned or migrated arbitrarily to another cloud when demand cycles increase. Moreover, cloud applications may have varying resource requirements during different phases of their lifecycle. Consequently, designing effective cloud orchestration techniques to cope with large-scale heterogeneous cloud environments remains a deeply challenging problem.

In this article, we propose a comprehensive analysis framework to effectively explore, assess, contrast, and compare the variety of resource orchestration techniques. Previous surveys mostly focused on specific aspects and appear fragmented. Topics include configuration management [Delaet et al. 2010], monitoring [Bauman et al. 2015], security and assurance [Ardagna et al. 2015; Huang and et al. 2015; Roy et al. 2015], energy efficiency [Mastelic et al. 2014], adaptability [Singh and Chana 2015; Zhan et al. 2015a], Quality of Service (QoS) and Service Level Agreements (SLAs) [Hani et al. 2015], as well as software architectures for cloud-based systems [Chauhan et al. 2016] and interoperability concerns [Toosi et al. 2014; Lewis et al. 2013]. A preliminary overview of cloud orchestration tools are also presented in Khoshkbarforoushha et al. [2016] and, similarly, an overview on cloud meta-models in Bergmayr et al. [2015]. Nonetheless, there remain significant shortfalls in both the complementarity and breadth of understanding within-cloud orchestration techniques. While previous efforts have produced encouraging and useful results, they are limited in scope with only a more "broad" significance. In contrast, we present a holistic and comprehensive framework. We propose a taxonomy that is much more exhaustive with additional (sub-)dimensions that contribute to an "in-depth" analysis over a mixture of techniques from both industry and academia. This is vital to understand the strengths and challenges and building blocks, in terms of concepts, models, languages, techniques and tools, and paves the way towards the next generation of cloud systems. To date, this level of investigation has received little attention, and this article aims to alleviate this gap.

We present an extensive survey in cloud resources orchestration. After introducing the necessary background (Section 2), we propose our taxonomy for understanding, analyzing, and comparing cloud resource orchestration techniques (Section 3). We also discuss related work and the positioning of our taxonomy versus existing attempts. Our taxonomy sets out a framework of dimensions (resources, orchestration capabilities, user types, runtime environment, and knowledge reuse), which we discuss progressively in Sections 4–8. We then apply the taxonomy to analyze a set of methodically chosen cloud resource orchestration tools and research prototypes and identify several open research issues based on the technical gaps identified during the analysis (Section 9). Finally, we offer concluding remarks and directions for future study.

2. CLOUD RESOURCE ORCHESTRATION

Consumers of cloud resources, human or software, typically have diverse requirements (e.g., storage capacity, access rules, etc.). Moreover, a single cloud resource often cannot provide all the necessary capabilities. Consider an Hypertext Transfer Protocol (HTTP) server, application runtime, and database, composed together to formulate a typical Web application deployment platform. The composition of dependent resources may require additional and complex configuration changes. For instance, a secured communication channel may be initialized between the application runtime and database by opening Internet Protocol (IP) ports and enforcing access rules (e.g., firewall rules). Furthermore, deployed resources produce events (e.g., application server started, database server crashed), which need to be monitored so necessary actions can be taken. To reason about this process, we introduce the notion of *Cloud Resource Lifecycle*, which aims to categorize orchestration tasks over the different phases in the typical lifespan of a cloud resource.

2.1. Cloud Resource Lifecycle

In much the same way practitioners have abstracted the lifecycle model, for example, in the case of software engineering artifacts [Larman and Basili 2003] or Business Process Management (BPM) [Dumas et al. 2013], we propose a similar lifecycle model suited for cloud resource artifacts. In essence, this model consists of the following phases: *(1) Selection,* consumers select required resources; *(2) Configuration,* resource description attributes are specified as well as relationships; *(3) Deployment,* cloud resources are instantiated; *(4) Monitoring,* resources are monitored to ensure they conform with QoS and SLAs; and *(5) Control,* resources are dynamically (re-)configured to alleviate violations or whenever there are changes in requirements.

At Section 5.1, we present a much more thorough description of these lifecycle phases together with relevant examples.

2.2. Cloud Resource Orchestration Services and Operations

To manage cloud resources over the lifecycle phases, various services and processes are used to *select*, *describe*, *configure*, *deploy*, *monitor*, and *control* cloud resources. We refer to the term *Cloud Resource Orchestration* to denote such processes and services. From the consumers' perspective, the function of orchestration systems are to bind resources and operations (e.g., deploy, monitor, scale-out), thereby providing an abstraction layer that shifts the focus from the underlying resource infrastructure to available orchestration services and resource management [Wang et al. 2012]. Cloud resource orchestration systems implement a service-oriented model, enabling consumers to satisfy their application requirements by utilizing resources from cloud environments. In this manner, the overall goal of cloud resource orchestration is to ensure successful hosting and delivery of applications by meeting the QoS objectives of consumers.

In Figure 1, we devise a reference architecture for cloud resource orchestration systems. In the following, we categorize processes and services involved in cloud resource orchestration based on their functionalities vis-à-vis this reference model.

—***Resource Provisioning Layer.*** Some services and tools merely offer the most basic operations to *create*, *reconfigure*, and *delete* cloud resources. Such services and tools are built on a *resource description model*—a meta-model that allows us to describe resource configurations. For example, AWS Command Line Interface (CLI) [AWS 2013b] provides a range of provisioning services for every resource that they support. One such service offers operations (e.g., create, start, stop, delete, clone, attach storage volumes) to provision EC2 virtual machines [AWS 2013a].

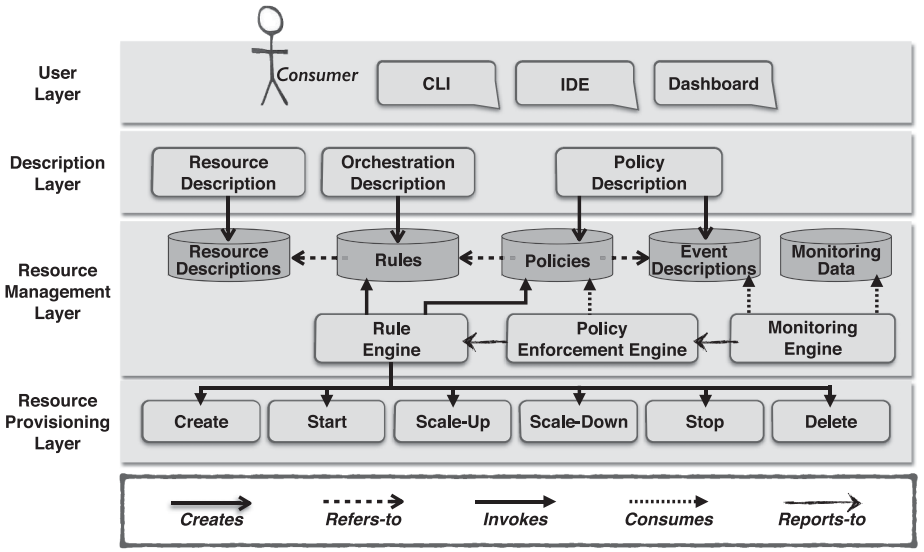

Fig. 1. Reference architecture for cloudresource orchestration.

—***Resource Management Layer.*** Effective automation of cloud resource *management* is imperative, as otherwise consumers are forced to manually build management logic over basic low-level operations offered by the *Resource Provisioning Layer*. For instance, automating a complex management task such as throughput-based Web application scaling in Amazon Web Services (AWS) requires (i) a monitoring engine (e.g., AWS CloudWatch [CloudWatch 2013]), (ii) a policy enforcement engine (e.g., AWS Auto Scaling [Auto Scaling 2015]), and (iii) a rule engine (e.g., Opscode Chef [Sabharwal 2014]). The monitoring engine collects throughput metrics from Web application servers and thereby publishes events to a Policy Enforcement Engine (PEE). Based on the captured metrics, the PEE determines what decisions to make (e.g., replicate the Web application into multiple instances). The PEE invokes the rule engine to execute orchestration processes (e.g., clone, deploy, and notify the HTTP load balancer about new instances). In some cases, execution may be delegated to a process engine, which coordinates the required scaling by leveraging operations at the *Resource Provisioning Layer*. Furthermore, services such as AWS Marketplace [Marketplace 2012] offer consumers to *discover*, *create*, *curate*, and *share* knowledge about resource provisioning and management as reusable artifacts.

—***Description Layer.*** This refers to languages and models to represent configuration, deployment, monitoring, and control tasks of cloud resources, typically as (i) resource descriptions, (ii) orchestration processes/rules, and/or (iii) policies.

  —*Resource Description*s define the *configuration* information of resources, as well as their relationships. For example, in AWS OpsWorks, a collection of Web application components (e.g., database, application engine, HTTP load balancer) and relationships can be defined via JavaScript Object Notation (JSON) notation [Amazon 2015b; Rosner 2013].

  —*Orchestration Description*s describes the "behavioral" aspects (i.e., *control* and *re-configuration*) of the cloud resources. There are *declarative* approaches (e.g., CloudFormation provided by AWS or Heat provided by OpenStack) or *imperative* approaches that are based on processes (i.e., workflows). In some cases, consumers explicitly define deployment and/or configuration rules. For example, AWS

OpsWorks provides a language with a set of pre-defined lifecycle events (e.g., setup, configure, deploy, undeploy, shutdown) that may be associated to orchestration actions. Other approaches, often based on workflows, require no explicit rules, albeit the *Rule-Engine* may delegate to a process-engine.

—*Policy Description*s endow resources with dynamic control behaviors, for example, defining load-based policies to scale Web applications. Such a policy allows us to instantiate new application engines when the average Central Processing Unit (CPU) utilization exceeds 95% and stop application engines when their average CPU load falls below 40%.

—***User Layer.*** Cloud resource consumers (e.g., system admin or app developers) may interact with services of the other layers. CLIs, Software Development Kits (SDKs), APIs and Integrated Development Environments (e.g., AWS CLI, AWS Java SDK, AWS REST API, and VisualOps) expose services to manipulate cloud resource descriptions, orchestration rules, and policies [AWS 2013b, 2015a, 2015b; VisualOps 2015]. Dashboards allows for interactions using user-friendly abstractions. For instance, Amazon CloudWatch [CloudWatch 2013]) represents *monitoring Data* using format pre-built User Interface (UI) components and charts.

3. CLOUD RESOURCE ORCHESTRATION TAXONOMY

To offer systematic analysis, we introduce our taxonomy as depicted in Figure 2. We identify the main dimensions and common building blocks that characterize cloud resource orchestration techniques and available solutions. The taxonomy is a result of our own research efforts, experiences from industry, extensive literature reviews in related areas, as well as experiments with various services and tools.

Earlier, we discussed *what* is meant by "cloud resources orchestration." Based on the identified taxonomy, we now focus *how* such orchestration can be described, deployed, and provisioned—independently of specific technologies or target solutions. Accordingly, we identify *five* main dimensions to characterize cloud resource orchestration techniques, which in turn are split into various sub-dimensions. A portion of this analysis also requires figuring out how a specific dimension may affect others (e.g., which resource access methods are suitable for which user categories).

(1) ***Resources***. This dimension identifies the formalisms that are offered for representing cloud resources. We further consider *what* resources are supported and *how* resources are modeled, represented, and accessed (refer to Section 4).
(2) ***Orchestration Capabilities***. This consist of actions and processes to manage orchestration tasks. We further divide this dimension into sub-dimensions and look at orchestration actions, paradigms, automation strategies, and theoretical foundations (refer to Section 5). (We summarize cross-cutting concerns at Appendix B.)
(3) ***User Type***. In our analysis, we identified three categories of users who have different roles in managing cloud resources (refer to Section 6).
(4) ***Runtime Environment***. We identify three relevant sub-dimensions: (i) Virtualization technique, (ii) Execution model, and (iii) Target environment.
Virtualization technique refers to *how* physical resources are abstracted to simplify their consumption. The *Execution* model refers to *how* cloud resources are deployed, monitored, and controlled in a distributed environment. *Target environment* identifies different deployment models such as public, private, and federated/hybrid cloud environments (refer to Section 7).
(5) ***Knowledge Reuse***. Productivity may be further enhanced through supportive reuse capabilities of existing orchestration knowledge. Users may implement and share orchestration knowledge as reusable software artifacts (e.g., resource descriptions, orchestration rules). We identify two sub-dimensions of knowledge reuse: (i) Reused Artifact and (ii) Reuse Technique (refer to Section 8).

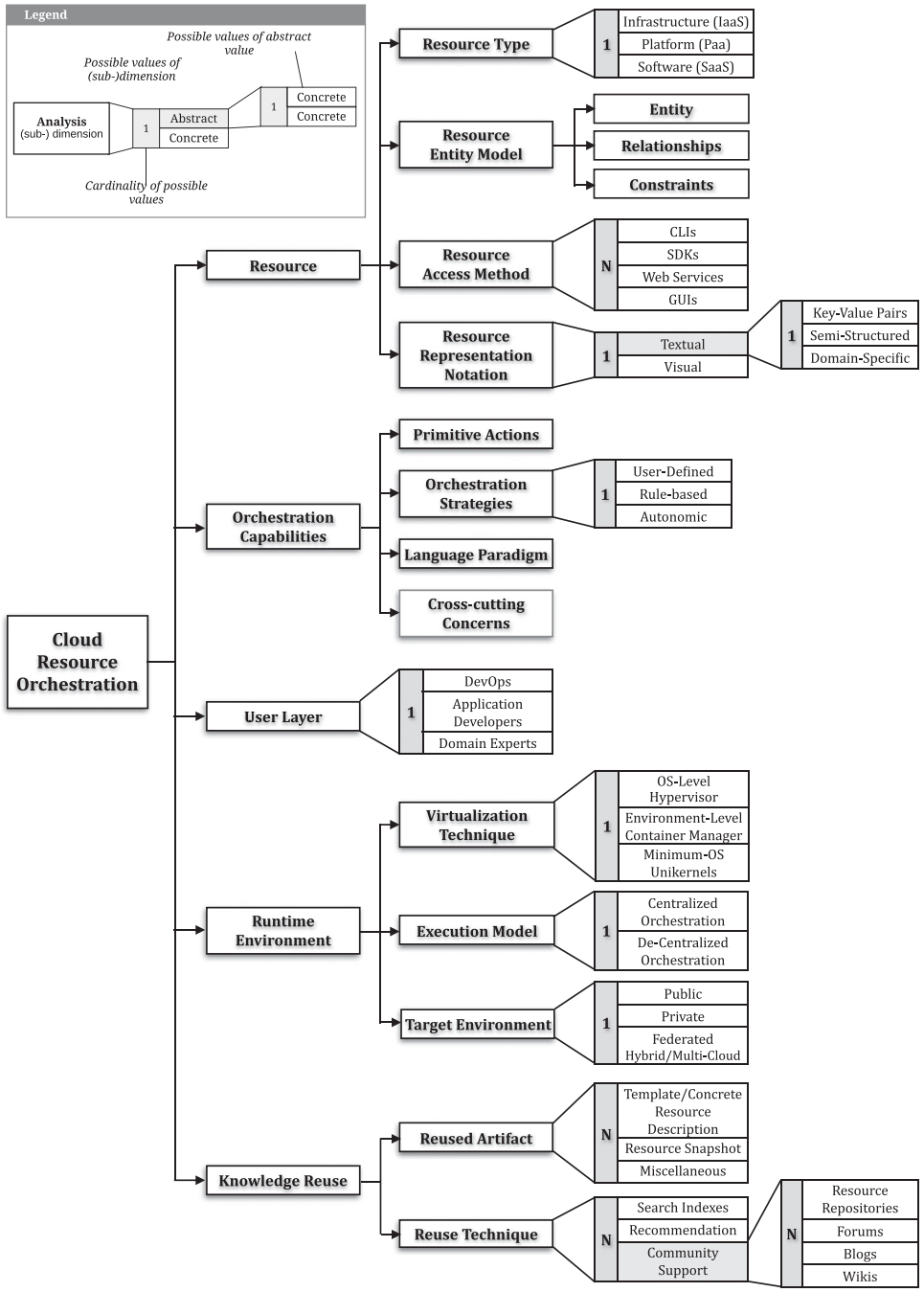

Fig. 2. A taxonomy in cloud resource orchestration.

3.1. Related Work: Positioning versus Existing Taxonomies

A holistic taxonomy and comprehensive framework for in-depth analysis has not yet been addressed comparable to this article (also refer to Table V of Appendix A).

High-level cloud computing taxonomies have been defined by OpenCrowd [OpenCrowd 2010] and leading industry vendors (Intel [Intel-Corporation 2015], Oracle [Oracle-Corporation 2011], and Cisco [Cisco-Systems-Inc. 2011]), although their applicability thus far is limited to understanding their (i) product strategies and business capabilities, (ii) key cloud product suppliers, and (iii) cloud computing strategies. In contrast, our taxonomy addresses technical resource orchestration concerns for all resource types (Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS)) across all orchestration layers (*Resource Provisioning*, *Resource Management*, *Description*, and *User Layers*); refer to Figure 1.

Rimal et al. [2009] proposed high-level concepts for some (sub-)dimensions (relative to Figure 2): Resource type (layered cloud service organization), Runtime Environment (visualization technique and multi-cloud interoperability), and Orchestration Strategies (load-balancing and fault-tolerance). However, while we have applied our taxonomy to a wide variety of academic and industrial approaches, Rimal et al. only analyzed offerings by vendors such as AWS, GoGrid, Flexiscale, and Azure.

Forrester [Ried et al. 2010] introduced a market-oriented taxonomy, and early cloud taxonomies by Hoefer et al. [2010] and Laird [2008] took a very simplistic view. These works failed to incorporate technical concerns and dependencies across multiple orchestration layers, which is clearly a novel contribution of our article.

Chana et al. [Singh and Chana 2016] devised an autonomic (self-* properties) resource management taxonomy that covers aspects related to runtime QoS management of cloud-hosted applications, including (i) monitoring, (ii) fault tolerance, (iii) workload consolidation, and (iv) scheduling objective function and QoS metrics. On the other hand, Toosi et al. [2014] presented cloud resource management with a focus on application portability across federated public/private cloud data centers. Similarly to Singh and Chana [2016], the taxonomy presented in Toosi et al. [2014] is limited to QoS and SLA management aspects (autonomic scheduling, portability, monitoring, security, cross-cloud communication). Similarly, Zhang et al. [Zhan et al. 2015b] presented a taxonomy to help in understanding and analyzing the application of Evolutionary Computing (EC) techniques for formulating application scheduling heuristics (i.e., *Orchestration Strategies*). Although the taxonomies presented in Singh and Chana [2016], Toosi et al. [2014], and Zhan et al. [2015b] are very detailed from the perspective of application QoS management (which evidently draws a parallel with *Runtime Environment* dimension and *Orchestration Strategies* sub-dimension in Figure 2), they failed to cover other important dimensions and sub-dimensions related to holistic cloud resource orchestration process, such as *User Layer*, *Knowledge Reuse*, *Resource Access Method*, *Resource Representation Notation*, *Language Paradigm*, and so on. Refer to Figure 2.

Different standardization bodies have also proposed reference cloud computing architectures, including National Institute for Standards and Technology (NIST) [Liu et al. 2011c], Distributed Management Task Force (DMTF) [DMTF 2010], Cloud Security Alliance [CSA 2011], and the Internet Engineering Task Force [Khasnabish et al. 2011]. Although these reference architectures aid in general understanding of the cloud computing model, they are not based on a systematic taxonomy. They hence lack in-depth technical concepts (i.e., dimensions and sub-dimensions) required to understand the holistic nature of cloud resource orchestration processes, which is clearly a novel contribution of our proposed taxonomy.

Several recent articles [Beloglazov et al. 2011; Shuja et al. 2014; Hameed et al. 2016] have introduced taxonomies related to energy-efficient scheduling (an instance

of *Orchestration Strategy*) of applications on cloud data centers. In particular, Shuja et al. focuses on issues related to designing cloud datacenter servers and network architectures such that they are optimized for energy efficiency and sustainability. Our taxonomy does not explicitly focus on energy efficiency but rather proposes a holistic taxonomy that covers end-to-end lifecycle aspects of cloud resource orchestration processes.

Attempts to define an ontological model to understand basic cloud resource types and their entity models, dependencies, and orchestration operations has been undertaken by several authors [Moscato et al. 2011; Fang et al. 2015; Zhang et al. 2012b; Dukaric and Juric 2013]. For example, EU mOSAIC's project [Moscato et al. 2011] proposed an ontology that lets application developers understand basic resource types and their entity models, dependencies, and configurations in a multi-cloud environments. On the other hand, Fang et al. [2015] proposed the Service Access and Manipulation Operation Specification (SAMOS) ontology that models entities (concepts), supported orchestration operations across SaaS, PaaS, and IaaS layers, and entity-to-entity relationships. Similarly, Dukaric and Juric [2013] propose an ontological taxonomy to characterize IaaS resource types and related orchestration operations. The taxonomy is structured around seven orchestration layers: core service layer, support layer, value-added services, control layer, management layer, security layer, and resource abstraction. Finally, EU Cloud4SOA project [Kamateri et al. 2013] proposed an ontological model to express relationship and dependencies between PaaS offerings across different cloud providers that share the same virtualization technology. Although these ontological models identify the necessary information related to cloud resource entity models and their relationships with other entities, in contrast to our work, these ontological taxonomy models lack the focus on other important taxonomy dimensions (*Orchestration Capabilities*, *User Layer*, *Knowledge Reuse*, *Runtime Environment*), which are highly mandatory for understanding the holistic nature of the cloud orchestration process.

In another strand of cloud computing research, authors have developed a taxonomy [Fatema et al. 2014; Alhamazani et al. 2015] for cloud monitoring (a type of runtime orchestration operation). However, these works have considered the monitoring problem in silos—and have failed to cover other important resource orchestration dimensions, such as *Orchestration Capabilities*, *Resource*, *RunTime Environment*, *User Layer*, and *Knowledge Reuse*. We present an encompassing general-purpose framework focused on all essential, interdependent dimensions of the cloud resource orchestration process.

4. RESOURCES

4.1. Resource Types

Cloud providers enable virtualition through three categories of resources, namely *Infrastructure*, *Platform*, and *Software*-as-a-Service.

—***Infrastructure.*** Infrastructure resources represent processing, storage, network, and hosting environments [Bittman 2011; Armbrust and et al. 2010; Wang et al. 2012; Thrash 2010; Ranjan et al. 2015]. Providers include VMWare vSphere, OpenStack, AWS EC2 CLI, Google Cloud Platform, OpenNebula, Eucalyptus, CohesiveFT, CloudStack, and Rackspace [Lowe 2011; OpenStack.org 2015a; AWS 2013a; Platform 2015; Networks 2016; Project 2016; Development 2016; CloudStack 2016; Cali 2013]. However, some providers do not support all types of infrastructure resources. For example, Rackspace allows us to describe virtual machines (VM), associate storage volumes, and create communication channels among VMs. On the other hand, Juju [Ubuntu 2013] only supports provisioning Ubuntu-based VMs and does not support storage or network resources.

—***Platform.*** Platform resources provide software development tools, middleware, SDKs, and/or APIs. It also supports runtime environments, such as content delivery networks, mobile application, and big-data platforms; all facilitate coding and deploying software resources. Providers include AWS OpsWorks, AWS CloudFormation, Google App Engine, Cloud Foundry, Ubuntu Juju, Puppet, Chef, Ansible, Heroku, EngineYard, CloudBees, and nitrous.io [Rosner 2013; Amazon 2011; Google 2015b; Cloud-Foundry 2016; Ubuntu 2013; Middleton et al. 2013; Engine Yard 2016; CloudBees 2016; Nitrous 2013; Puppet 2015]. For example, Heroku provides language runtimes such as Java, Ruby, and Node.js.

—***Software.*** Software resources are applications (i.e., Web or mobile) for a service-based software delivery model [Cusumano 2010; Ranjan et al. 2015]. For example, *Salesforce.com*[1] provides pay-per-use Customer Relationship Management. Software resources are the most abundant type of resources compared to Platform or Infrastructure resources [Gartner 2014].

4.2. Resource Entity Model

We propose the notion of a *Resource Entity Model* to represent the structure of cloud resources and their relationships. This implies a high-level resource model, which we represent as a graph, whose nodes and edges correspond to cloud *Resource Entities* and their *Relationships*, respectively [Chen 1976], as well as any related *Constraints*.

4.2.1. Resource Entity Types. An entity type describes properties of cloud resources via a set of attributes (e.g., key-value pairs), thus characterizing their possible runtime instances. For example, a VM provided by AWS EC2 [Services 2015a] includes attributes such as the number of CPUs, storage and memory capacity, the operating system, and access rules. System administrators specify values prior to deploying, and afterwards it may include additional attributes (e.g., instance ID, public IP address, and launch time).

Resource entities can be further categorized as *Elementary* or *Composite*. An elementary resource does not rely on any other resources while acting as the primary building blocks of composite resources. A composite resource is an umbrella structure that brings together other elementary and composite resources to describe a larger cloud resource, for example, an E-Learning platform that consists of an artifact management service and student identity management service to support 100 students.

Resource entities may be described at various levels of granularity. For example, Puppet [Kanies 2006] orchestrates resources within a single physical/virtual machine. Primary resource entities are thus fine grained, for example, *file*, *sshkey*, and *package* [Labs 2015c]. Coarse-grained resources such as app-engines (e.g., Node.js runtime) are composed of fine-grained resources. In contrast, Juju [Ubuntu 2013] orchestrates resources deployed across multiple machines. Juju provides *Charms*, which represent high-level services (e.g., Node.js runtimes, Hadoop clusters) as primary resource entities.

Most orchestration techniques only support describing resources of a specific provider [Konstantinou et al. 2009; Wittern et al. 2014; AWS 2013b]. On the other hand, others such as Topology and Orchestration Specification for Cloud Applications (TOSCA), ModaClouds, and CloudBase provide *cross-provider Resource Entities* that are portable across different providers [Binz et al. 2013; Ardagna and et al. 2012; Weerasiri et al. 2015]. Orchestration techniques that support *cross-provider* resources (e.g., *ComputeService* in JCloud) are often intended for configuration and management of federated or hybrid cloud resources [Foundation 2014b; Elmroth and Larsson 2009; Villegas and et al. 2012].

[1]http://www.salesforce.com

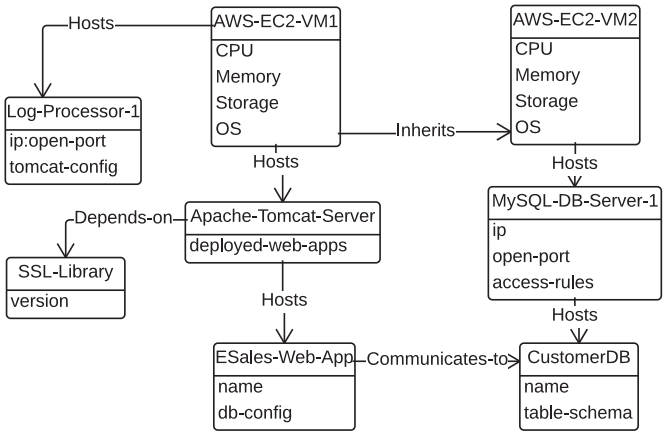

Fig. 3. Resource entities and relationships of a Web application.

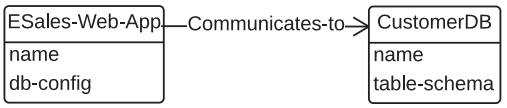

Fig. 4. Communication relationship between a Web application and database.

4.2.2. Resource Relationships. A *Relationship* denotes a link between two *Resource Entities*. The relationship constructs can be further annotated with key-value pairs to describe the properties of the respective relationship.

In circumstances where an orchestration technique does not support the explicit descriptions of relationships, composite resources may in fact become inconsistent when orchestrating two related component resources. Consider a Web application, such as a **L**inux software stack, including an **A**pache HTTP server, **M**ySQL database server, and **P**HP application engine (LAMP) suite [Lawton 2005]. When the associated database server is migrated to a new IP address, this means that the relevant configuration attributes held at the application engine should also be updated. This is required to maintain successful communication between the application engine and database server. However, if the orchestration technique does not support explicit relationships, this implies system administrators may need to manually update the relevant attributes (or employ other third-party tools such as shell scripts). These alternatives are error prone and may also cause unnecessary overheads.

Relationships are established between a *provider* and *consumer* resource entities, where the *provider* offers a certain capability for the *consumer*. Figure 3 exemplifies relationships of a typical Web application named *ESales-Web-App* with other resources. This Web application is "hosted" in *Apache-Tomcat-Server* and "communicates" data provided by *CustomerDB*, which is "hosted" in *MySQL-DB-Server-1*. *Apache-Tomcat-Server* and *MySQL-DB-Server-1* are "hosted" in *AWS-EC2-VM1* and *AWS-EC2-VM2*, respectively.

We identify the following types of relationships between cloud resources:

(1) *Communication Relationship.* Denotes the exchange data. For example, TOSCA 1.0 [OASIS 2013] provides a relationship type called *ConnectsTo*, for example, between an application and its associated database (refer to Figure 4). TOSCA 1.0, thereby, interprets description attributes (e.g., communication protocol) of the relationship and constructs a channel between the relevant resources.

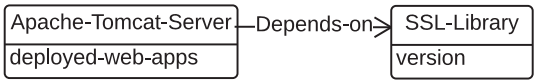

Fig. 5. *Apache-Tomcat-Server* depends on *SSL-Library*.

Fig. 6. Inheritance relationships between *Docker Images*.

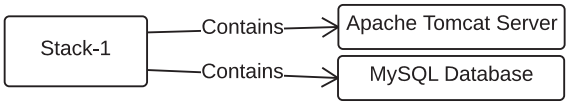

Fig. 7. Containment relationships within an *OpsWorks Stack*.

(2) *Dependency Relationship.* Associate a given resource with other supporting resources that are required for successful operation. For example, a Web application server depends on a Secure Socket Layer (SSL) library (e.g., OpenSSL) to encrypt and communicate data with other resources, such as a database server (refer to Figure 5). TOSCA 1.0 provides a relationship type called *DependsOn*. In Ubuntu Juju [Ubuntu 2013], relationships are described as resource attributes that specify whether a given resource *provides* or *requires* a particular capability to/from another resource. For instance, MySQL DB *provides* a data source, whereas a Web application *requires* a data source). System administrators are able to create these relationships during deployment.

(3) *Inheritance Relationship.* Denotes when the *provider*'s attribute values are inherited by the *consumer*. However, the *consumer* resource is permitted to override the inherited attribute values to enable customizations. In other words, inheritance relationships are a convenient way of configuring attributes of a resource entity by *reusing* attribute values of another resource entity. For example, to describe a new Web application, which is to be installed on Apache Web server and Ubuntu Operating System: An application developer may simply inherit an existing *Web Application* resource with a similar configuration—all relevant attributes are inherited (refer to Figure 6). Similarly, in Figure 3, an Inheritance relationship is set up from *AWS-EC2-VM1* to *AWS-EC2-VM2*. This relationship enforces VM2 to include the same version of operating system described VM1.

(4) *Containment Relationship.* Denotes a parent-child relationship in which orchestration actions on a parent automatically trigger actions on all children. In practice, containment relationships are used to conveniently orchestrate a set of related resource entities together. For example, AWS OpsWorks [Rosner 2013] provides a resource entity type called *Stack*. It represents a Web application and may contain a set of child entities that are required to build a Web application, such as an Apache Tomcat Server and Mini SQL (MSQL) database (refer to Figure 7). When the *Stack* entity is deleted, all children are deleted automatically.

(5) *Hosting Relationship.* Enforces deployment of the *consumer* within the *provider* resource. This is useful when multiple component resources need to be deployed within a single component resource. For example, a log-file processor and an application server need to be deployed within a single VM, as the log-file processor needs the local file system access to read application server logs (see Figures 8 and 3). For example, Ubuntu Juju [Ubuntu 2013] enables users to specify the

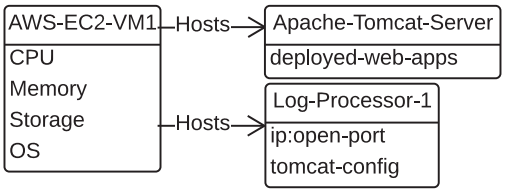

Fig. 8. Web Application Server and Log Processor (hosted in one VM).

infrastructure resource provider (e.g., AWS, HP-Cloud, Windows Azure) that will be used to deploy platform resources. Similarly, TOSCA 1.0 [OASIS 2013] supports a hosting relationship called *HostedIn*, where the deployment engine interprets the relationships and resolves which resource is to be hosted into which resource.

4.2.3. Constraints. In some circumstances, it may be necessary to restrict the type of resource *entities* or *relationships*. For instance, AWS does not allow creating EC2 VMs with arbitrary amounts of CPU, memory, and storage. Instead, a set of VM types (e.g., micro, medium, large) are provided that are optimized for different use cases (e.g., low-traffic Web applications, large databases) [Amazon 2015c]. From a non-technical standpoint, having a constrained set of VM types allows providers to maintain simpler billing policies. Similarly, constraints may restrict resource relationships. For example, an AWS EC2 VM must be configured with a 64-bit CPU to install a 64-bit Operating System (OS) on the particular AWS-EC2 VM.

For resource entities, constraints are specified by restricting the possible values of attributes. For relationships, constraints are specified using *cardinality* and *participants*. Consider a Ubuntu OS installed within an AWS EC2 VM. Here a one-to-one relationship exists, given that neither the Ubuntu OS can exist within more than one VM, nor can multiple OSs be installed with a VM simultaneously, whereas one-to-many relationships may exists between a cluster of HTTP Web servers and their load balancer. Sometimes the cardinality may be arbitrary, for example, Ubuntu Juju allows users to specify maximum and minimum numbers of consumers (e.g., Web application) that may create relationships with a provider (e.g., Web application engine) [Juju 2015b].

We also identify two further orthogonal sub-categories of the *Participants* relationship, namely (i) Inter-Vendor and (ii) Vendor-Specific relationships.

(a) *Inter-Vendor relationships.* In some cases, relationships are permitted between two participating resources from different orchestration vendors. For example, Google App Engine [Google 2015b] allows users to deploy applications on a Google Compute VM or as a Docker container. DevOps are therefore allowed to associate infrastructure and platform resources across these different vendors.
(b) *Vendor-Specific relationships.* Some providers do not permit relationships between other vendors and in some cases even restrict between different resource types (i.e., infrastructure, platform, and software). For example, DotCloud [dotCloud 2015] only permits a composition of platform resources (e.g., databases, application engines) on top of a specific infrastructure resource (e.g., AWS EC2). DevOps are therefore not allowed to configure or reconfigure the infrastructure resources. On the other hand, CA-AppLogic allows users to specify which platforms resources are to be deployed on which infrastructure resources.

Furthermore, standards such as Open Cloud Computing Interface (OCCI) [Metsch et al. 2010] may be imposed as a means for providing semantics over resources, defining the type of a given entity, describing interdependencies between various entities and defining operating characteristics on them. The goal being to facilitate extensibility and interoperability.

4.3. Resource Access

Over the years, software interfaces have evolved, offering various designs to cater to the different capabilities of diverse users. Similarly, in the context of cloud orchestration, we have identified four types of interfaces [Khoshkbarforoushha et al. 2016].

4.3.1. CLIs. CLIs offer a fixed set of commands, each of which includes a specified set of input, output, and error parameters. For example, the AWS CLI [AWS 2013b] suite allows users to configure, deploy, and control cloud resources, such as VMs, data storage, and load balancers. As shown in Code 1, the "run-instances" command allows DevOps to deploy a specified number of VMs in the AWS public cloud infrastructure. The input parameters describes the VM configuration and the number of VMs to be launched in terms of key-value pairs. The output of the command execution is a JSON-based description of the resultant deployment.

```
1 aws ec2 run-instances --imageId = 1a2b3c4d
2                       --count = 1
3                       --instanceType = t1.micro
4                       --keyName = MyKeyPair
```

Code 1: AWS CLI command to deploy VMs

4.3.2. SDKs. AWS provides SDKs for a wide range of languages (e.g., Java, PHP, .NET, and Ruby). For example, DevOps may download the Java-based SDK and thereby write Java applications to configure and deploy cloud resources in AWS cloud infrastructure (refer to Code 2). While CLIs are intended for system administrators with less application development skills, SDKs are intended for those with expertise in particular programming languages.

```
1 RunInstancesRequest runInstancesRequest = new RunInstancesRequest();
2
3 runInstancesRequest.withImageId("1a2b3c4d")
4     .withInstanceType("t1.micro")
5     .withMinCount(1)
6     .withMaxCount(1)
7     .withKeyName("MyKeyPair"); /* specifying attributes of the VM */
```

Code 2: Java Syntax in AWS SDK to deploy a VM

4.3.3. APIs. Compared to SDKs, APIs provide language-independent interfaces for orchestration capabilities that can be accessed by software applications, typically over HTTP. For example, Rackspace provides a RESTful API [Rackspace 2015] to configure, deploy, and control cloud resources such as VMs, load balancers, and databases. Moreover, techniques also exist to simplify access and integration with APIs [Barukh and Benatallah 2013a, 2013b].

4.3.4. Graphical User Interfaces (GUIs). GUIs comprise visual constructs to interact with orchestration services. For example, StackEngine, Panamax, and Shipyard provide Web based GUIs to configure, deploy, monitor, and replicate Docker containers [StackEngine 2015; CenturyLink 2015; Shipyard 2015]. CA-AppLogic provides a desktop-based GUI to manage software appliances in a private cloud infrastructure [AppLogic 2015]. Some other advanced GUIs, such as Puppet Enterprise Console and VisualOps, provide dashboards that generate reports such as bar charts and maps (e.g., visualize the number of failed and running VMs during past 30 days) [Labs 2015b; VisualOps 2015].

4.4. Resource Representation Notation

Notations for representing resources and their relationships may consist of textual and/or visual constructs. We identified three classes, namely textual, visual, and hybrid (a mix of textual and visual) notations.

4.4.1. Textual Notations. We distinguish between three variations of textual notations:

(1) *Key-value.* This consists of a set of unique keys (or attributes) that characterize cloud resources. A schema is also provided that defines the range of possible values for particular keys. This type of notation is commonly used among providers that offer CLIs. For example, the command for creating VMs in AWS CLI [AWS 2013b] expects DevOps to provide values for keys such as "image-id" and "instance-type" to describe the VMs to be created (refer to Code 1).
(2) *Semi-structured.* Semi-structured data formats, such as YAML Ain't Markup Language (YAML), eXtensible Markup Language (XML), and JSON, offer a structuring mechanisms for organization of key-value pairs. They define markers to separate and enforce hierarchies among different key-value pairs. Compared to other notations, semi-structured notations are better suited for representing complex cloud resource configurations. For example, DotCloud follows YAML-based resource descriptions, which include both basic and composite configuration attributes [dotCloud 2015]. Each branch in the root level represents a basic cloud resource configuration (e.g., Java VM, node.js engine, PHP engine).
(3) *Domain-specific.* Docker [Turnbull 2014] allows a domain-specific notation known as *Dockerfiles* to be written. Each specifies the configuration parameters of a particular cloud resource (refer to Code 3). DevOps may also describe a composition of a set of cloud resources using a file named *docker-compose.yml* (refer to Code 4).

```
1 FROM python:2.7                          # this application is based on python 2.7
2 ADD . /code                              # commands to install the python application
3 WORKDIR /code
4 RUN pip install -r requirements.txt
5 CMD python app.py                        # command to start the python application
```

Code 3: *Dockerfile* of a Python Web application

```
1 web:                      # configuration parameters for the Web application
2   build: .
3   ports:   - "5000:5000"
4   volumes: - .:/code
5   links:   - redis        # setting up the communication link with the database
6 redis:                    # configuration parameters for the database
7   image: redis
```

Code 4: *docker-compose.yml* of Web application and *Redis* database

4.4.2. Visual Notations. Visual programming languages abstract technical details with "visual symbols" and "graphical notations" [Chignell et al. 2010]. For example, CA AppLogic Cloud Platform [AppLogic 2015] provides a notation, with a catalog of constructs that represents elementary platform resources (e.g., databases, routers) and other visual constructs to describe composite platform resources (e.g., Web applications). Figure 9 depicts a Web application, composed of an HTTP Gateway (i.e., *IN*), Web application Server (i.e., *WEB5*), and a network-attached storage (i.e., *NAS*) [Technologies 2013].

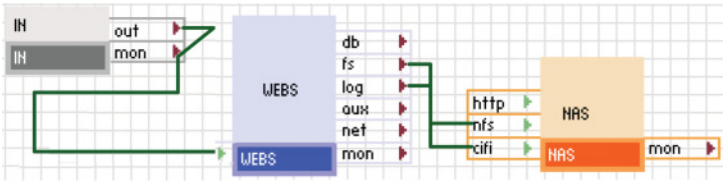

Fig. 9. Visual notation in CA-Applogic for a Web application.

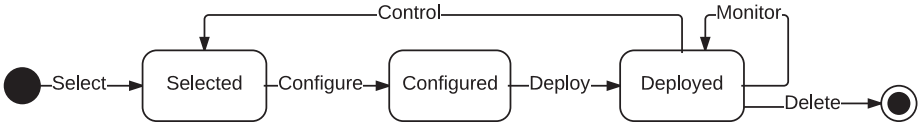

Fig. 10. State transitions of the cloud resource life cycle.

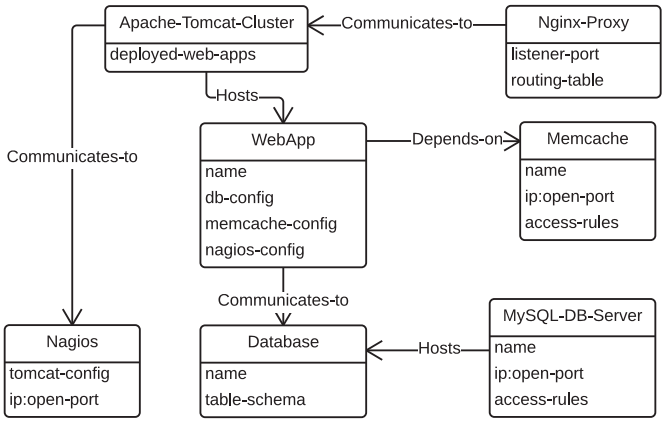

Fig. 11. A Composite Resource Infrastructure for a Web application.

5. RESOURCE ORCHESTRATION CAPABILITIES

Implementing orchestration processes can vary from a simple sequence of primitive actions to complex processes.

5.1. Primitive Actions

Primitive actions are depicted as state transitions in the state chart in Figure 10. These actions are based on the typical cloud resource lifecycle model that we introduced in Section 2.1. We may explain these actions in the context of a real-world scenario: Figure 11 depicts a composite cloud resource for a Web-app runtime (using the *Resource Entity Model* presented at Section 4.2). This includes an Apache-Tomcat application engine cluster with Web-apps deployed at each node. Nginx is a reverse proxy to distribute incoming traffic to Web-apps at each node. Nagios is a monitoring service to observe the throughput of node clusters. The MySQL database server maintains data. MemCache is configured as a caching service, which improves the performance of database calls.

(1) *Select.* DevOps first need to *select* resources that satisfy their requirements. For instance, if a database is required, a viable set of database providers are evaluated and selected based on both functional (e.g., storage capacity and type of the database) and non-functional requirements (e.g., availability and cost per unit).

For example, Bitnami [2015] provides a selection service where consumers search and select cloud resources based on intended task category (e.g., project management, Web application) and target deployment environment (e.g., personal desktops, VMWare vShpere private cloud, AWS public cloud).

(2) *Configure.* Next, resources are configured by defining the expected properties and relationships. For example, in AWS OpsWorks [Rosner 2013], consumers choose required resources (e.g., MySQL DB server) and provide configuration attributes (e.g., Database (DB) server with 5GB of capacity and running on port 3306) to define an expected runtime behavior. These descriptions are then submitted to the *Resource Provisioning Layer* (refer to Section 2.2) to create the desired cloud resources.

(3) *Deploy.* Deployment involves interpreting descriptions and preparing resources into an operation and consumption-ready state. For example, administrators may use AWS-RDS API [Amazon 2015a] to provision a MySQL DB-server where the database in Figure 11 is created. System administrators may then configure its tables manually or via an ad hoc script (refer to Code 5).

```
1 #provisioning and starting the database server with configuration attributes
2 rds-create-db-instance mysqlDatabase -s 10 -c db.m1.large -e mysql -
      u admin -p password
3 #creating the database and tables via an SQL script
4 mysql -h mysqlDatabase.rds.amazonaws.com -P 3306 -u admin -p < ''
      dbAndTableCreationScript.sql''
```

Code 5: Linux shell commands to deploy a database server in AWS-RDS

Once the component resources are constructed, relationships must be created. For example, the necessary ports and access rules should be set up within the Apache Tomcat application engine cluster and MySQL DB-server such that requests and responses can be sent and received between the application and DB (see Figure 11).

(4) *Monitor.* Once the cloud resources are operational, DevOps must monitor to ensure resources are continuously operating according to requirements. For example, a Tomcat application engine configured to be operational on 24x7, should be monitored to check that it responds to incoming requests continuously. If found otherwise, it implies the SLA has been violated between the provider and consumer. Nagios is a monitoring engine [Barth 2008] to specify events to be monitored (via a command definition) and receive notifications (e.g., via email). For example, Code 6 checks whether the Tomcat application engine is running, and a notifications may be sent (as defined in *Contact definition*), which sends an email to *admin@abc.com* when the engine is not running.

(5) *Control.* When cloud resources are monitored and found to be not operating according to the configured attributes, DevOps or automated processes may take necessary *control actions* to recover from the situation. For example, Figure 12 depicts the orchestration logic that scales an Apache Tomcat application engine cluster when the network throughput becomes less than 95%.

5.2. Orchestration Strategies

We classify cloud resource orchestration techniques in accordance with their level of sophistication. Less-sophisticated techniques require more human interventions (and vice versa), particularly to orchestrate resources in response to dynamic changes.

5.2.1. User-Defined Orchestration Strategies. User-defined orchestration strategies are the most basic form of implementing cloud processes. DevOps implement orchestration processes as ad hoc scripts that exploit only a set of primitive actions supported by

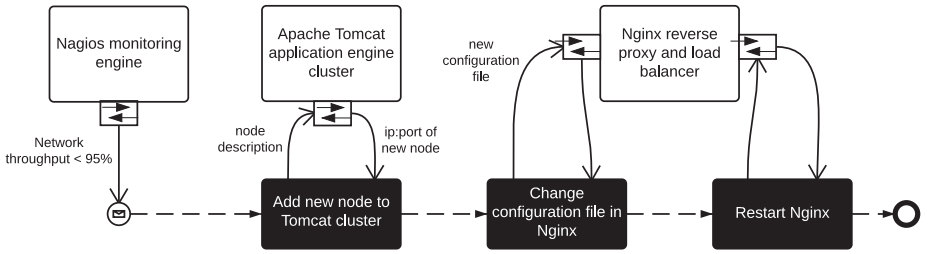

Fig. 12. Orchestration Workflow (in black and bold) for scaling up an Apache Tomcat Application Engine Cluster.

```
# Host definition (where the Tomcat application engine is hosted)
define host{
  use linux-server
  host_name AWS-EC2-Host-1
  address 201.168.1.3
  contact_groups admins
}
# Service definition of Tomcat application engine
define service{
  use generic-service
  service_description Tomcat-Engine-1
  hostgroup_name AWS-EC2-Host-1
  contact_groups admins
  check_command check_Tomcat
}
# Command definition for the health check of Tomcat application engine
define command{
  command_name  check_Tomcat
  command_line  ps -ef | grep tomcat; if[ $? -gt 0 ];
                then echo "Tomcat Pass"; else echo "Tomcat Fail"; fi
}
# Contact definition
define contact{
  contact_name admins
  email adminabc.com
```

Code 6: Nagios syntax to monitor a Tomcat application engine

a particular orchestration language. Existing cloud resource orchestration techniques typically rely on User-defined orchestration strategies, written in general-purpose or domain-specific scripting languages [Ranjan et al. 2015; Liu et al. 2011a; Lu et al. 2013; Zeng et al. 2004]. For example, Chef recipes [Chef 2015] follows a domain-specific scripting language that extends Ruby to specify orchestration actions such as configuration, deployment, and deletion of resources (refer to Code 7).

```
file "/etc/config.txt" do          # location of the file
  owner 'root'
  group 'root'
  mode '0755'                      # access permissions
  action :create
end
```

Code 7: A Chef Recipe representing the Configuration and Deployment of a File

However, scaling up or down cloud resources in dynamic environments via *User-defined* orchestration processes leads to a costly and inflexible solution. This adds significant complexity, insists on extensive programming effort, calls for multiple and continuous patches, and perpetuates closed-cloud solutions. To alleviate this somewhat, tools such as *Juju GUI*, *OpenTOSCA*, and *VisualOps* provide *visual abstractions* to describe deployment workflows and resource topologies [Ubuntu 2013; Binz et al. 2013; VisualOps 2015]. For example, *AWS Management Console*, *VisualOps*, *CA AppLogic*, and other cloud resource management tools provide control features such as *restarting*, *scaling*, and *migration* [VisualOps 2015]. Moreover, monitoring tools such as *Nagios* and *CloudFielder* can allow DevOps to define SLA, detect anomalies, and notify about SLA violations.

5.2.2. Rule-Based Orchestration Strategies. Some providers define a reactive rule-based language in addition to primitive actions. This means that Event-Condition-Action (ECA) rules may be specified based on pre-defined events or patterns [Michelson 2006]. When events are detected, the specified actions are auto-triggered by the orchestration engine. For example, AWS OpsWorks [Rosner 2013] supports five event types (i.e., *setup*, *install*, *deploy*, *undeploy*, and *shutdown*). Actions consist of Chef recipes [Chef 2015]. Several research initiatives have adopted this strategy: Chapman et al. [2012], Zhang et al. [2011], and Zabolotnyi et al. [2015]. Code 8 shows an elasticity rule to dynamically deploy new VMs as the "number of jobs awaiting execution increases" [Chapman et al. 2012].

```
1  <ElasticityRule name="AdjustClusterSizeUp">
2    <Trigger>
3      <TimeConstraint unit="ms">5000</TimeConstraint>      //Event
4      <Expression>                                          //Condition
5        (@uk.ucl.condor.schedd.queuesize /
6        (@uk.ucl.condor.exec.instances.size +1) > 4) &&
7        (@uk.ucl.condor.exec.instances.size < 16 )
8      </Expression>
9    </Trigger>
10   <Action run="deployVM(uk.ucl.condor.exec.ref)" />      //Action
11 </ElasticityRule>
```

Code 8: Elasticity Rule to Scale VMs (adapted from Chapman et al. 2012)

5.2.3. Autonomic Orchestration Strategies. With the expanding complexity of cloud-based systems, orchestration tasks become too cumbersome to be carried out largely with human-assisted techniques including user-defined and rule-based strategies. Autonomic orchestration strategies, the highest level of sophistication, refer to self-managing features of cloud resources [Toosi et al. 2014; Singh and Chana 2015]. For instance, endowing resources with self-management capabilities has the potential to enhance high availability of cloud resources, for example, through dynamic (re-)configuration, to maintain the expected quality of service in the presence of faults, variable environmental conditions, and changes in user requirements [Singh and Chana 2015; Zhan et al. 2015a; Cheng and Garlan 2012; Yuan et al. 2014]. For example, an autonomic orchestration process automatically scales up or down running applications by analyzing the recent resource consumption statistics. This implies that orchestration techniques intelligently make certain decisions when managing cloud resources without taking any instructions from users [Parashar and Hariri 2005].

It should be noted that, in terms of self-managing cloud resources services, technology is still in the early stages. CometCloud is an example of effort in this direction [Kim and Parashar 2011]. Supported features include budget-, deadline-,

and workload-based deployment of cloud-based applications. Other efforts include automatic re-configuration of resources to meet evolving application resource requirements and QoS [Zhan et al. 2015a; Singh and Chana 2015; Toosi et al. 2014; Ye et al. 2016]. For example, *Fuzzy BPM-aware Auto-Scaler* scales up or down VMs based on Key Performance Indicators of deployed business processes within the VMs [Schulte et al. 2015; Mamdani 1974]. Other research initiatives trigger orchestration actions and processes based on analysing user requirements (e.g., SLAs), end-user context (e.g., geolocations and device configurations), and environmental properties (e.g., unit cost per resource, processing speed of VMs) [Wei and Blake 2013; Gravier et al. 2015; Menzel et al. 2015; Zhang et al. 2012a; Zabolotnyi et al. 2014; Fang et al. 2012].

Learning-based methods, based on historical or simulated data, have been applied to support autonomic cloud resources orchestration. Xu et al., Islam et al., A Self-Adaptive Prediction system (ASAP), and Online Resource Management Decision Support System (ORMDSS) propose neural-network models that are trained using different workload scenarios to determine an optimal or near-optimal configuration of VMs and software appliances [Xu et al. 2012; Islam et al. 2012; Jiang et al. 2011; Ramezani et al. 2013]. Antonescu et al. propose a learning-based technique to migrate and provision cloud-based mobile services based on the mobility of users [Antonescu et al. 2013].

Various heuristic-based resource allocation and migration algorithms have also been proposed to support autonomic orchestration of cloud resources [Mishra et al. 2012; Pandey et al. 2010; Beloglazov et al. 2012; Iqbal et al. 2011]. Some of these are based on pre-defined policies that determine which type of VMs should be provisioned to which data centers, while optimizing the energy consumption [Beloglazov et al. 2012].

We identify several other methods that include formally defined abstractions for specifying automated resource orchestration, namely *Closures*, *Promises*, and *Aspects*.

Closures encapsulate orchestration commands as black boxes; this aids in reducing management complexity and costs [Couch et al. 2003; Burgess and Couch 2006]. Its behavior can be thought of as the sum of its transactions with the outside world, such that each output from a *closure* is a function of all input received. Inputs take the form of events and streams. *Closures* are adopted in CFEngine [Burgess and College 1995].

Promises model the way cloud resources commit to certain behaviors [Burgess and Couch 2006; Bergstra and Burgess 2014]. It allows cloud resources to become more autonomous and self-sufficient in dynamic environments. In CFEngine, *Promises* are implemented as policies that modify resources, such as those in non-conforming states that are transformed into conforming states [Burgess and College 1995]. Effectively, this approach immunizes cloud resources against potential deterioration by continuously repairing those non-conforming. *Promises* are also idempotent; they will do nothing unless non-conformity is discovered. This technique have been applied to verification and knowledge management of cloud orchestration [Burgess 2011, 2009].

Aspects are an abstraction for organizing *Promises* into distributed bundles and constellation [Burgess 2007]. *Aspects* are introduced over *Promises* to describe complex orchestrations that need to be dealt with by multiple *Promises* simultaneously.

5.3. Language Paradigm

Language paradigm is an "approach of programming based on a coherent set of principles and practices" that determines its "suitability for solving certain types of problems" [Van Roy et al. 2009]. We have identified the following language paradigms used in cloud orchestration systems (e.g., Puppet, Chef, Juju, Docker, SmartFrog, AWS OpsWorks) and research efforts [Cui et al. 2013; Chieu and at al. 2010; Goldsack et al. 2009; Delaet et al. 2010; Konstantinou et al. 2009; Wilson 2009].

5.3.1. Imperative Programming. We have identified three sub-categories as follows:

—*Script-Based.* DevOps widely adopt scripting languages (e.g., JavaScript, Python, Bash) to implement cloud resource orchestration processes. Providers that use this method include *Docker* and *Vagrant* [Turnbull 2014; Hashimoto 2013].
—*Flow-Based Programming.* The primitive constructs of flow-based orchestration languages are data-flow and control-flow connectors. This approach is based on service composition and business process modeling languages (e.g., Business Process Execution Language (BPEL), Business Process Model and Notation (BPMN) [OMG 2011; Juric and Weerasiri 2014]. *BPMN4TOSCA* [Kopp et al. 2012] (which include four BPMN extensions) and *CloudBase* [Weerasiri et al. 2015] extend BPMN to implement orchestration processes of cloud applications.
—*Rule-Based Programming.* ECA rules are specified by associating a sequence of configuration, deployment, or re-configuration actions for each of possible events [Ubuntu 2013]. Code 8 exemplifies an ECA rule that dynamically deploys new VMs as the "number of jobs awaiting execution increases" [Chapman et al. 2012].

5.3.2. Declarative Programming. We have identified three sub-categories as follows:

—*Markup Languages.* Plush is a tool to deploy, monitor, and control distributed software applications. It advocates an XML-based language to model and deploy software components [Albrecht and et al. 2011].
—*Query-based.* Query-based orchestration languages model cloud resources as structured data (e.g., tables, graphs, trees) and provide actions (e.g., create, read, update, and delete) for processing structured data. Liu et al. [2011a, 2011b] represents cloud resources as a treelike data structure and provide declarative primitives to create, delete, and update cloud resources.
—*Constraint Programming.* Constraint programming enables automatic generation of cloud resource configurations from declarative constraint specifications [Danninger 2015; Sawyer et al. 2012]. For example, *CFEngine* automatically determines the steps required to create and update resource configurations by analyzing constraint specifications and recent changes within the operating environment [Burgess and College 1995].

6. USER TYPES

We identify *three* types of users involved in orchestrating cloud resources, namely *DevOps, Application Developers*, and *Domain-Experts*. DevOps is an emerging role to consolidate application developers and system administrators. To effectively manage resources, complex orchestration processes needs to be carried out (e.g., setting up an application testing environment, testing application updates, migrating the tested environment to the production, and scaling the production environment based on usage patterns). DevOps are responsible for optimizing and automating those orchestration processes, which improves the quality of software development and continuous delivery processes. The traditional application developer's role is also important, including the use of cloud for resources management. However, not all application developers are DevOps (which implies the distinction of roles in our analysis). For example, developers may just be responsible to write Java code that are later deployed on Heroku.

A domain expert is specialized in a specific domain (e.g., biologists, teachers) and may use cloud resources for their work processes. For example, a Harvard University lecturer for an introduction to computer science course [Malan 2015] may create a virtual machine named *CS50 Appliance 19* [CS50 2015], which includes all software required by students to develop, test, deploy, and execute code. Domain experts have very little or no programming expertise. For this reason, it is imperative that

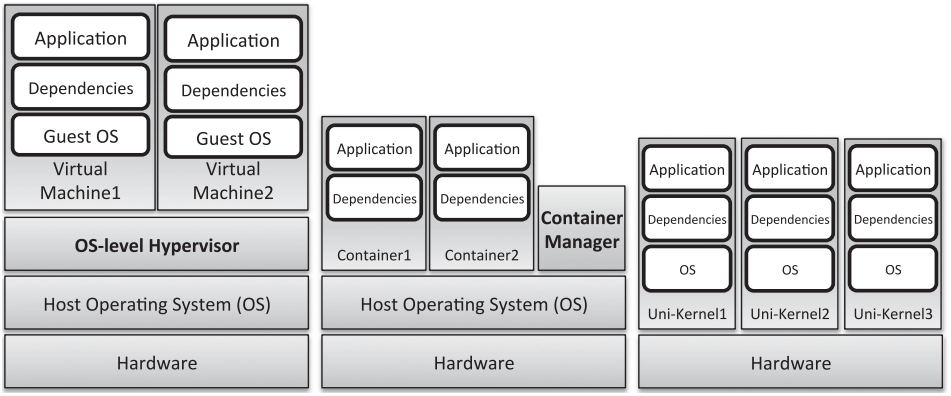

Fig. 13. OS-level hypervisor (left) vs. container (middle) vs. unikernels (right).

domain experts are provided with end-user-oriented or domain-specific orchestration languages. *RightScale Self-Service* allows domain experts to automatically provision and orchestrate *Software* resources based on high-level policies [RightScale 2016]. It also provides a Web-based portal, whereby domain experts can deploy resources by specifying non-technical resource attributes and assess their financial costs.

7. RUNTIME ENVIRONMENT

The runtime environment for cloud orchestration relies on *three* orthogonal concerns: (a) virtualization technique, (b) execution model, and (c) target environment.

7.1. Virtualization Technique

Virtualization is the key technique that transforms cloud resource descriptions into concrete resources; it provisions hardware and software constituents without upfront capital expenditure [Chapman et al. 2012]. It addresses three main concerns: (a) performance isolation, (b) data isolation, and (c) execution isolation [Gupta et al. 2006]. *Data interference* is the unintended data sharing (e.g., file systems) across different resources. *Execution interference* is the effect on the runtime state (e.g., failures) of one resource to another resource. *Performance interference* is the influence of the performance of one resource to another, where both share the same underlying resources.

We discuss *two* types of virtualization techniques that are commonly adopted by cloud resource orchestration techniques:

—*OS-level Hypervisor*. The virtualization component runs on top of a host operating system, with VMs installed with a guest operating system (Figure 13 (left)). The virtualization component accesses a shared pool of resources (e.g., memory, CPU, and system calls) through the host operating system and partitions resources across operating systems. For example, the AWS EC2 service uses an extended version of Xen as their OS-level hypervisor to provision EC2 virtual machines.

—*Environment-Level Container Manager*. The virtualization component runs on top of the kernel of a host operating system, similarly to OS-level hypervisors. In contrast, however, the hardware layer is not virtualized but uses features of the operating system kernel to create lightweight virtualized operating system environments, that is, containers (Figure 13 (right)). For example, Linux containers are built by leveraging *cgroups* and *namespace* features of the Linux kernel [LinuxContainers.org 2015; Rosen 2013]. Environment-level containers do not require installing separate

guest operating systems on each container—they share the hardware layer and host operating system kernel layer across all containers. This is a resource isolation mechanism with little overhead compared to OS-level hypervisors. Docker is a container manager on top of the Linux OS [Turnbull 2014].

—*Minimum-OS Unikernels.* The virtualized component implements the bare minimum operating system's kernel libraries just enough to support successful execution of the component. Unlike containers, Unikernels-based (e.g., MirageOS, Rump Kernel, Clive) virtualized components [Oliver 2015] do not share OS kernel libraries with other containerized instances. By presenting the ability to compose application components that are not only lightweight while having the security features of OS-level hypervisors, Unikernels have begun to emerge as a viable alternative to overcome the current security concerns relevant to container-based (hypervisor-free) approaches. Moreover, application developers have explicit control over core security areas, as they can choose which kernel library to package or turn off by default. Similarly to containers, unikernels are much lightweight [Oliver 2015; Darvell 2016] as compared to traditional, full-blown VMs that are hosted on top of OS-level hypervisors (e.g., *Hyper-V*, *Xen*).

7.2. Execution Model

The execution model refers to *how* a particular orchestration process distributes and performs tasks. We identify *two* main types of execution models:

—*Centralized Orchestration.* In this model, the execution manager performs all the tasks of an orchestration process. If tasks are dispersed across a set of machines within a distributed environment, then the centralized manager directly issues commands to perform the orchestration. For example, VMWare vSphere [Lowe 2011] is a virtual machine management tool that creates and manages VMs on top of a single host machine. Ansible performs as a central manager, which directly issues orchestration commands via the Secure Shell (SSH) protocol; such that the issued commands are received by remote machines [Mohaan and Raithatha 2014].

—*De-Centralized Orchestration.* In this model, all participating machines are required to install an *agent* supplied by the orchestration provider. During execution, tasks are delegated to the agent—which is thereby responsible to perform the actual orchestration tasks. Agents are only aware of their delegated tasks and not about tasks assigned to other agents. For example, *Puppet* supports agent-based orchestration in which there is a central server that stores orchestration processes. Agent machines periodically poll the central server for orchestration tasks and perform those tasks. Puppet follows a model based on *Promises* (refer to Section 5.2.3) to avoid potential inconsistencies in autonomous and de-centralized orchestrations [Bergstra and Burgess 2008]. Kirschnick et al. [2012] propose a peer-to-peer architecture, a highly scalable and fault-tolerant architecture with no central orchestration server, to automatically deploy software components across a pool of virtual machines.

7.3. Target Environment

7.3.1. Public Cloud. Public cloud providers, such as AWS, provide a range of orchestration techniques (e.g., *AWS Command Line Interface*, *AWS CloudFormation*, *AWS OpsWorks*), each of which suits different types of users (e.g., system administrators, DevOps, developers) to configure, deploy, and control cloud resources. Alternatively, there are third-party cloud resource orchestration techniques that provide plug-ins to integrate with public cloud providers. For example VisualOps provides a graphical interface to configure and visualize VMs deployed across different regions (e.g., Europe, Australiasia) in the AWS environment [VisualOps 2015].

7.3.2. Private Cloud. Private cloud resource providers, such as *VMWare* and *OpenStack*, offer *VMWare vSphere* and *Heat*, respectively, to configure and manage virtual machines within a private network [Lowe 2011; OpenStack.org 2015b]. Additionally, third-party tooling such as *Juju*, *Ansible*, *Chef*, and *Puppet* [Ubuntu 2013; Mohaan and Raithatha 2014; Sabharwal 2014; Puppet 2015; Kanies 2006] support resource configuration, deployment, and control in OpenStack-based private cloud deployments.

7.3.3. Federated Cloud. Tools for orchestrating federated cloud resources have been introduced in both research and industry [Toosi et al. 2014; Hajjat et al. 2011]. They either (a) define a unified cloud resource orchestration language that must be conformed to by all participating providers [Weerasiri et al. 2015; Wettinger and et al. 2014] or (b) provide a pluggable architecture that interprets different orchestration languages of participating providers [Wettinger et al. 2014; Weerasiri et al. 2015].

For example, TOSCA is an open standard for representing and orchestrating cloud resources [OASIS 2013; Binz et al. 2013]. It describes a federated cloud resource using a *Service Template*. This template captures the topology of component resources and sets a plan for orchestrating those resources.

Techniques for capturing a unified representation, as well as enabling orchestration of cloud resources among diverse providers, have been studied by research [Moscato et al. 2011; Smit et al. 2013] and implemented as language libraries [Foundation 2014c; Tidwell 2009; Foundation 2014a, 2015c]. On the other hand, *Ansible* provides a suite of distinct language modules, each of which publishes an orchestration interface for a specific resource type offered by a particular resource provider (e.g., AWS, Rackspace, Azure, VMWare) [Ansible 2015]. *Ansible* is thus able to implement scripts by reusing a set of modules to model and orchestrate federated cloud resources.

Federated Cloud is a key factor to facilitate switching providers, that is, avoiding vendor lock-in and optimizing cost-to-performance tradeoffs. Recognizing this competitive edge, cloud resource orchestration tools and techniques are attempting to expand their capability to compose, deploy, and manage applications across multiple cloud providers. Nonetheless, the federated cloud model is nontrivial to design, and it is difficult to implement generic resource orchestrators that can work with various providers. We (and others [Toosi et al. 2014]) believe that Federated Cloud can be realized either using *Multi-Cloud* or *Hybrid Cloud* abstractions. While we do note other opinions that view slight distinctions between Multi-Cloud and Federation [Grozev and Buyya 2014; Petcu 2014], we uphold the previously stated view.

When considering Federated Cloud environments, security and regulatory compliance requirements must be considered, as they vary considerably across providers.

Resource orchestration in Multi-Cloud environments refer to transparently integrating IaaS and PaaS resources offered by multiple public cloud providers as part of a single application composition. Administrators can manage and automate application movement and the communication among resources hosted in different clouds. Resource orchestrators provided by *RightScale*, *EngineYard*, and *CohesiveFT* support application deployment across multiple clouds (e.g., *Amazon Web Services*, *Microsoft Azure*, *HP Cloud*) by implementing an adapter layer (e.g., based on APIs such as *Apache jclouds*) that hides the low-level technical complexity (e.g., hypervisor type, authentication, authorization, networking) of heterogeneous, multiple clouds from high-level application composition (configuration, monitoring, runtime adaptation).

On the other hand, resource orchestration in Hybrid Cloud environments refer to transparently shedding excess application workload to one or more external public clouds, when private cloud resources are not able to cope with the demand for computing capacity spikes. The major advantage of Hybrid Cloud is that an organization only pays when needed for extra resource capacity. Resource orchestrators from commercial

PaaS providers, such as *CloudSwitch* (which supports bursting of in-house workload to AWS and recently acquired by Verizon), and EU FP7 Projects, such as *OPTIMIS* and *SeaClouds*, support deployment of applications in Hybrid Cloud environments.

8. KNOWLEDGE REUSE

Knowledge reuse frameworks are based on four main pillars: (a) knowledge representation, (b) knowledge acquisition, (c) knowledge curation, and (d) knowledge discovery. Knowledge *representation* techniques are presented in Section 8.1. We then discuss various methods used for knowledge *acquisition*, *curation*, and *discovery* in Section 8.2.

8.1. Reused Artifact

An artifact may be *atomic* (e.g., a resource description or orchestration rule) or *composite*, including multiple interrelated elements (e.g, deployment workflow). Reuse artifacts can be distinguished as *template* or *concrete*. Concrete artifacts are fully developed orchestration solutions. Template artifacts are generalized solutions that need manual adaptations (e.g., initializing configuration parameters) before reuse. Considering the above, we have identified the following variety of reuse artifacts.

8.1.1. Resource Description Templates. Most enterprise-ready cloud orchestration providers support both concrete and template resource description repositories for knowledge reuse. For example, Google Container Engine, Docker, and Juju offer cloud knowledge repositories (i.e., Google Container Registry, Docker Hub, and Juju Charm Store) [Google 2015a; Docker 2015a; Canonical 2015; Services 2015b]. Docker Hub enables sharing and reusing resource descriptions by means of *Docker Images*, which represent resource deployment descriptions (e.g., mongoDB database, nginx reverse proxy server) with the required dependencies. Docker Hub may be used to discover, configure, and deploy existing *Images*. Template *Images* are associated with a set of configuration parameters (e.g., access credentials of a database server Image) that are initialized by users before the deployment, while concrete *Images* have pre-initialized configuration parameters.

8.1.2. Resource Snapshots. A snapshot of a cloud resource includes not just its description but also a specific runtime state (e.g., deployed and started application server). In contrast to reusing resource description templates, snapshots additionally embed information about the execution of the orchestration process. For example, *Snaps* in *terminal.com* and VMware Snapshots provide resource snapshots [Cloudlabs-Inc. 2015; VMware 2015]. Users of *terminal.com* (e.g., application developers) may specify, deploy, and share *Snaps* with other users (e.g., QA engineers, system administrators) who may test, monitor, and control those *Snaps*.

8.1.3. Miscellaneous. DevOps create and publish DockerFiles, which are textual resource descriptions of *Docker Images*; they may then be shared on code repositories such as GitHub. Instructions for how to configure and deploy the specific DockerFile into a *Docker Container* may also be shared. However, these instructions can only be interpreted by humans (not machine read).

8.2. Reuse Techniques

Given an artifact for reuse, it is imperative to identify different techniques that can be applied in practice to enable its reuse. We identify the following *three* categories.

8.2.1. Search Indexes. Ansible, Puppet, and Chef provide search indexes based on resource description attributes (e.g, artifact name, owner, version, and created date) [Mohaan and Raithatha 2014; Puppet 2015; Sabharwal 2014]. This assumes that users

know the exact (or nearly exact) attributes values to query for potential artifacts to reuse. There are more advanced search indexes (e.g., Bitnami) that accept query inputs such as intended task category (e.g., project management) and target deployment environment (e.g., AWS EC2 public cloud, VMWare vSphere private cloud).

8.2.2. Recommendation. This approach implies proactively suggesting a set of potential artifacts to facilitate the orchestration. Compared to search indexes, recommended artifacts are suggested based on application profiles, usage histories, contexts, and so on [Resnick and Varian 1997; Weerasiri and Benatallah 2015; Zhang et al. 2012c]. For example, AWS Marketplace suggests virtual appliances based on users' ratings and comments. Additionally when users choose a particular virtual appliance (e.g., http server), a list of related virtual appliances (e.g., http load balancer) is recommended that can be deployed along with the chosen appliance.

8.2.3. Community-Driven Techniques. Leveraging user-expertise to facilitate knowledge reuse is a popular choice among many enterprise-level cloud providers.

—*Resource Repositories.* As mentioned, online databases such as Docker Hub and AWS EC2 Container Registry act as Git-like version-control repositories [Docker 2015a; Services 2015b]. Some communities like Bitnami [Bitnami 2015] restrict all but authorized developers to register resource artifacts. Others, such as, Ubuntu Juju [Juju 2015a] and Puppet [Labs 2015a], implement strict curation policies (e.g., licensing, naming conventions, idempotency of orchestration rules) when sharing artifacts. Other communities, such as Docker Hub [Docker 2015a], do not enforce curation policies; they implement reputation schemes to collectively estimate quality and correctness of resource artifacts.

—*Forums, Blogs, and Wikis.* Forums allow users to post questions and ideas and receive targeted answers and comments from other users. For example, Puppet provides a forum for DevOps to post, query, answer, and rate questions. *Blogs* usually contain information authored by a single user or organization. For example, the Chef community posts blog articles about artifact development best practices, updates to the orchestration language, and other related news. *Wikis* are community-driven collaborative environments that are particularly useful for small teams to maintain documentation of cloud resources. For example, DevOps can keep track of the list of deployed VMs. DevOp may then update the wiki whenever they make any changes (e.g., installing software, operating system updates).

9. APPLYING THE TAXONOMY: EVALUATION OF CLOUD RESOURCE ORCHESTRATION TECHNIQUES

In consolidation of the foregoing discussion, we organize the analysis of state of the art by characterizing techniques and tools along the main dimensions of our taxonomy (as presented in Section 3). We include well-known enterprise tools and frameworks, as well as initiatives derived from a wide selection of research literature.

9.1. Selection Process

Careful consideration was applied in the selection of relevant tools for our analysis; this entailed several phases of investigation: Initially, 20 orchestration tools were chosen from a set heavily advocated by the DevOps community. We experimented with those tools to understand the main dimensions that are common among these tools. Based on our observations, we were able to derive the initial draft of our taxonomy. Furthermore, we derived analysis tables that summarized each tool according to the relevant dimensions that were identified as part of the initial taxonomy. We then chose a selection of research initiatives from leading, critically reviewed research

proceedings (research and demonstration tracks), magazines, and journal articles that were relevant to the domain from the year 2004 onwards. In particular, these included the following conferences: Cloud Computing (CLOUD), Cloud Engineering (IC2E), Service-Oriented Computing (ICSOC), Advanced Information Systems Engineering (CAiSE), Large Installation System Administration (LISA), Database Systems for Advanced Applications (DASFAA), Cooperative Information Systems (CoopIS), Cloud Computing and Services Science (CLOSER), and Utility and Cloud Computing (UCC), and in the following journals: *ACM Computing Surveys*, *ACM Transactions on Internet Technology*, *IEEE Internet Computing*, *IEEE Transactions on Network and Service Management*, *IEEE Transactions of Cloud Computing*, and *Journal of Systems and Software*. We analyzed these initiatives and further revised our taxonomy and comparison tables based on our findings.

Ultimately, 11 different cloud orchestration approaches were selected for analysis, namely AWS OpsWorks [Rosner 2013], AWS CloudFormation [Amazon 2011], VMWare vSphere [Lowe 2011], Heroku [Middleton et al. 2013], Puppet [Kanies 2006; Puppet 2015], Juju [Ubuntu 2013], Docker [Turnbull 2014], OpenTOSCA [Binz et al. 2013], CFEngine [Burgess and College 1995], Plush [Albrecht and et al. 2011], and SmartFrog [Goldsack et al. 2009].

9.2. Resources and User Types

Table I maps the selected orchestration techniques onto the *Resources* and *User Types* dimensions described in Sections 4 and 6. The supported resource types, access methods, and representation notations immensely influence the type of users. Therefore, to appreciate this correlation, we present our analysis of these two dimensions together.

Accordingly, by studying the characteristics relative to these two dimensions, we summarize our findings as follows:

—Eight of 11 approaches utilize *Domain-specific* representation notations; however, there are an assortment of other notations with the same or similar representation capabilities (as cited in Section 4.4). This underlines the factual assertion and suitability of domain-specific notations in the field of cloud resource orchestration.
—Ten of 11 approaches support CLI-based resources access; from which 7 also provide API-based access. Due to the fact that Linux- and Unix-based systems are managed via CLIs, the current DevOps community is heavily equipped with CLI-based system administration skills. To manage applications across public and private clouds, providing APIs and SDKs for programmatically accessing resources become an important requirement.
—Ten of 11 approaches support representing *platform* resources. In general, cloud resource orchestration vastly remains the prerogative of professional DevOps, although the adoption of end-user intuitive visual abstractions is emerging.

As a future direction, it will be vital to provide effective end-user-oriented representation capabilities. This will be complimented by diversifying techniques for *resource representations*, more specifically as follows:

—*End-User empowered declarative representation.* We identify *end-users* as an important and emerging category of *user-type* for orchestration techniques in the future. Accordingly, end-users should be able to easily and declaratively represent cloud resources, as well as access, configure, compose, and analyze simple yet powerful composite cloud resources. Currently, even sophisticated DevOps are often forced to resort to grasping different low-level resource access methods, and procedural language paradigms, to create and manage complex cloud resources.

Table I. The *Resource* and *User Type* Dimensions of the Selected Platforms

	Resource						
		Res. Representation	Res. Entity Model			Res. Access	User
	Res. Types	Notation	Entities	Relationships	Constraints	Method	Type
AWS OpsWorks	Platform Resources	Domain-specific notation based on Chef cookbooks	Support defining composite trees from component entities. Component entities are Web application components.	Support Containment relationship to compose a set of related resources required for a Web application	Constraints are allowed in entities as attributes and rules (e.g., auto-scaling rules in Layer)	Web based GUI, CLI, SDK, APIs	DevOps
AWS Cloud-Formation	Infrastructure and Platform resources	JSON	Support defining composite graphs from component entities	(1) Support Dependency relationships between resources. (2) Support Containment relationships to group all the related resources	(2) Local attributes in resource entities (2) Attributes can be defined to apply on all the resource entities	CLI, APIs	DevOps
VMWare vSphere	Virtual Machines	Visual	Top level resource is a ServiceInstance (a data center). ServiceInstance can be modeled a set of VMs that can be composed of component entities like network, alarm	(1) Support Dependency relationships between resources. (usually these relationships can be modeled between the component resources within VMs) (2) Support Containment relationships to group all the related VMs	Support attributes in resource entities to configure VMs	Desktop based GUI, CLI, APIs	System Admin-istrators
Heroku	Platform	Domain-specific	Support defining composite trees from component entities	(1) Support Containment relationships to group a set of Dynos that belong to a particular app. (2) Support Dependency relationships (e.g., pom.xml in java apps)	(1) Support attribute based constraints in resource entities. (2) Policies can be specified on particular entities (eg., At least one Web Dyno entity should exist in each App entity)	CLI, APIs	DevOps
Puppet	Platform Resources	Domain-specific	(1) Supports a graph of resource entities (2) Entity types include files, packages like resource that can be composed to model a machine (3) Top level Composite entity represent a Machine (Physical/Virtual)	(1) Support Dependency relationships that results the deployment behavior among resource entities (2) Hosting relationships to specify which resource entities should be deployed on which machines	Resource entity specific constraints are provided as attributes. Puppet also define a hierarchical structure to categorize resource entities such that constraints defined in parent are inherited to children	CLI, APIs, Web-based GUI	DevOps
Juju	Infrastructure and Platform resources	YAML	(1) Support a graph of resource entities	(2) Dependency relationships between Charms (e.g., require, provide interfaces) (2) Containment relationship (e.g., between Charms and the Provider) (3) Hosting relationship (e.g., between a service-unit and a machine/container)	Support entity and relationship specific constraints via attributes	CLI, Web-based GUI	DevOps
Docker	Platform Resources	Domain-specific	Support a graph of resource entities	(1) Communication relationships (2) Dependency relationships (3) Hosting relationship	Entity specific constraints via attributes	CLI, APIs	DevOps
OpenTOSCA	Infrastructure and Platform resources	Visual notation	Support a graph of resource entities	(1) Communication relationships (e.g., connect to) (2) Dependency relationships (e.g., depend on) (3) Hosting relationships (e.g., hosted in)	Entity and relation specific constraints via attributes	Web-based GUI	DevOps

(Continued)

Table I. Continued

	Resource						
		Res. Representation	Res. Entity Model			Res. Access	User
	Res. Types	Notation	Entities	Relationships	Constraints	Method	Type
CFEngine	Platform Resources	Domain-specific	Support a graph of resource entities	(1) Supports Dependency relationships (e.g., depends_on) (2) Support Containment relationship	Resource entity specific constraints are provided as attributes	CLI, APIs, Web-based GUI	DevOps
Plush	Platform Resources	XML	Support a graph of resource entities	(1) Support Dependency relationships (2) Support Containment relationships to group all the related resources	Entity specific constraints via attributes	CLI	DevOps
SmartFrog	Platform Resources	Domain-specific	Support a graph of resource entities	(1) Supports Inheritence and Containment relationship	Entity specific constraints via attributes	CLI	DevOps

—*Adoption of open-standards.* Furthermore, while it is still not prevalent, the adoption of open standards (e.g., TOSCA, OVF, OCF) to represent reuse artifacts [OASIS 2013; Crosby et al. 2009; Foundation 2015a] would significantly assist DevOps to build portable and interoperable configurations across different cloud providers. Accordingly, we believe any type of representation paradigm would benefit by adopting open standards.

9.3. Resource Orchestration Capabilities

Table II maps the selected orchestration techniques onto the *Resource Orchestration Capabilities* dimension described in Section 5. We summarize our findings as follows:

—Seven of 11 approaches support user-defined orchestration strategies. Four of 11 support rule-based orchestration. However, to the best of our knowledge, none of the industry tools fully support autonomic resources orchestration. This manifests an important need for continued research on effective, intuitive, and autonomic cloud resources orchestration processes.
—We also observed, there are only a few tools, such as *Juju GUI*, *OpenTOSCA*, and *VisualOps*, that provide *visual abstractions* to describe deployment workflows and resource topologies [Ubuntu 2013; Binz et al. 2013; VisualOps 2015]. For example, *AWS Management Console*, *VisualOps*, and *CA AppLogic* provide control features such as *restarting*, *scaling*, and *migration* [VisualOps 2015]. Moreover, monitoring tools such as *Nagios* and *CloudFielder* can allow DevOps to define SLA, detect anomalies, and notify about SLA violations. Nonetheless, even then there are drawbacks among these approaches, in that DevOps often have to switch between multiple tools for different aspects of the management lifecycle, which proves time consuming and cumbersome.
—We have noticed that cross-cutting concerns are reasonably addressed by research initiatives, and more so among enterprise-ready orchestration techniques. This is likely because their utilization in production environments require solutions that address issues such as security, portability, and/or fault tolerance.
—It is apparent that various orchestration techniques employ different language paradigms. However, based on our observations, there is not yet any predominant language widely adopted by the majority of cloud orchestration providers.

Accordingly, we identify the following main issues as future directions for *Orchestration Capabilities*:

—*State-machine-based models for elasticity management.* We envision *state machines* as a novel abstraction to dynamically represent and reason about elasticity-aware

Table II. The *Resource Orchestration Capabilities* Dimension of the Selected Platforms

	Resource Orchestration Capabilities			
	Primitive Actions	Orchestration Strategies	Language Paradigm	Cross-cutting Concerns
A WS OpsWorks	Create, Delete, Describe, Update actions are provided for each resource entity. Clone, Start, Stop, Reboot actions are offered for some entities (e.g., Stack, Instance). Global actions are also provided. (e.g., SetLoadBasedAutoScaling)	Rule-based processes	ECA rule based	Security rules (authorization, access protocols), SLA can be defined via auto-scaling and auto-healing rules
AWS Cloud-Formation	Create, Delete, Update, Describe, and Clone are the main actions provided	User-defined processes	Markup language	Security rules (authorization, access protocols), SLA can be defined via auto-scaling rules
VMWare vSphere	Provide a large amount of actions for each entitiy type. In general all these actions can be categorize into create, delete, update.	Rule-based processes	Markup language	Security rules (authentication, authorization), Portable VMs
Heroku	Create, update, scale, delete applications, viewLogs (useful for monitoring)	Autonomic and user-defined processes	Script based	Security rules (OAuth authorization)
Puppet	Create, update and delete resources	Mainly User-defined processes, but rule-based processes for few resources	Constraint Programming	Security rules (encryption, authentication, authorization)
Juju	Create and delete (Environment, VMs, and Charms, Services, Relationships between Charms), describe Environment, detect Events, update Charm	Rule-based processes	ECA rule based	Security rules (authentication, authorization)
Docker	Create and delete (Image, Container), share (Image), start, stop, restart (Container), update (Container)	User-defined processes	Script based	Security rules (authorization, access protocols)
OpenTOSCA	Create, update and delete (resources and relationships, attributes)	User-defined processes	Flow based	Portable resources
CFEngine	Create, update and delete resources	Rule-based processes	Constraint Programming	Security rules (encryption, authentication, authorization)
Plush	Create (environment and application)	User-defined processes	Markup based and Flow based	Not addressed
SmartFrog	Deploy, start and terminate	User-defined processes	Markup language	Not addressed

resource orchestration techniques. Instead of directly manipulating low-level interfaces and scripting orchestration rules over complex cloud services, state machines may reason about resource requirement states. States may also characterize application-specific resource requirements (e.g., CPU and storage usages), constraints in terms of costs, and other SLAs. Transitions between states are triggered when certain conditions are satisfied (e.g., a temporal event, workload increases beyond a certain threshold). Transitions thereby automatically trigger control actions to perform the desired (re-)configurations over resources to satisfy the requirements and constraints of target states.

—*Visual techniques for orchestrating cloud resources.* DevOps are faced with orchestrating large amounts of complex cloud resource configurations. This involves being

Table III. The *Knowledge Reuse* Dimension of the Selected Platforms

	Knowledge Reuse	
	Reused Artifact	Reuse Technique
AWS OpsWorks	Concrete and Template resource descriptions	Search index
AWS CloudFormation	Concrete and Template resource descriptions	Search index
VMWare vSphere	Portable Resource snapshots	Search index, Recommendations, Community-driven approaches (e.g., blogs)
Heroku	Concrete and Template resource descriptions	Not specified
Puppet	Concrete and Template resource descriptions	Community-driven search indexes
Juju	Concrete and Template resource descriptions and Miscellaneous	Community-driven search indexes
Docker	Concrete and Template resource descriptions and Miscellaneous	Community-driven search indexes
OpenTOSCA	Portable Concrete and Template resource descriptions	Not specified
CFEngine	Concrete and Template resource descriptions	Search index
Plush	Concrete and Template resource descriptions	Not specified
SmartFrog	Concrete and Template resource descriptions	Not specified

able to proficiently understand and analyze cloud resource attributes and relationships and make orchestration decisions on demand. We therefore believe that cloud orchestration should be endowed with *visual* techniques to configure, deploy, monitor, and control cloud resources. For example, a visual approach may allow DevOps to perform orchestration tasks, such as drag, drop, and connect pre-built component cloud resources, as well as deploy, monitor, and manage composite cloud resources. Our previous work has introduced such a model-driven notation, based on a user-friendly and familiar mindmap interface [Weerasiri et al. 2016]. Beneath the surface, techniques are applied to manage, monitor, and control cloud resource orchestrations by mapping to underlying frameworks, such as Docker.

9.4. Knowledge Reuse

Table III maps the selected orchestration techniques onto the *Knowledge Reuse* dimensions as described in Section 8. We summarize our findings as follows:

—Research initiatives for cloud orchestration techniques generally underestimate the reuse of orchestration knowledge. Comparatively, all of the enterprise-ready approaches we analyzed provide some form of knowledge. This observation asserts the utmost *practical* necessity and importance of knowledge- reuse for DevOps to build and orchestrate real-world cloud resources.
—Seven of 11 approaches employed *search indexes*—the most prominent knowledge discovery technique. Among other search methods, keyword-based search is widely used. Generally speaking, recommendation-based knowledge discovery techniques are promising, albeit most orchestration providers do not adopt this approach due to the complexity of implementation and maintenance of the accuracy of recommendations.
—Enterprise-ready approaches predominantly support *community-driven* knowledge archival and curation techniques. This is due to the vast amount and diversity of cloud resources that needs to be supported. For instance, in the absence of the crowd, providers would have to build and maintain a knowledge artifact repository on their own, which would clearly be unfeasible in practice.

Table IV. The *Runtime Environment* Dimension of the Selected Platforms

	Runtime Environment		
	Virtualization Technique	Execution Model	Target Environment
AWS OpsWorks	OS-level hypervisor	Centralized	Public Cloud
AWS OpsWorks	OS-level hypervisor	Centralized	Public Cloud
VMWare vSphere	OS-level hypervisor	Centralized	Private Cloud
Heroku	Environment-level Container manager	Centralized	Public Cloud
Puppet	Not relevant (only responsible for configuration management of resources rather virtualizing them)	De-centralized	Public or Private Cloud
Juju	OS-level hypervisor	Centralized	Public or Private Cloud
Docker	Environment-level Container manager	Centralized	Public or Private Cloud
OpenTOSCA	OS-level hypervisor	Centralized	Public or Private Cloud
CFEngine	Not relevant (only responsible for configuration management of resources rather virtualizing them)	De-centralized	Public or Private Cloud
Plush	Not relevant (only responsible for configuration management of resources rather virtualizing them)	Centralized	Private Cloud
SmartFrog	Not relevant (only responsible for configuration management of resources rather virtualizing them)	Centralized	Private Cloud

The discipline of *knowledge reuse* follows a prevailing direction, namely devising a unified representation and reuse mechanism over heterogenous artifacts. This is similar to what query languages offered for databases and, more recently, for processes, also known as "hybrid processes" [Barukh and Benatallah 2014]. Likewise, it is paramount to invest in a unified representation, configuration, and reuse strategy over heterogenous *cloud resource knowledge* for simplified and productive cloud orchestration. Central to this, we prepose the concept of *Orchestration Knowledge Graphs*, where common low-level orchestration logic can be abstracted, incrementally curated, and thereby reused by DevOps. The type of knowledge captured can be organized into dimensions, including Intended tasks, Resource providers, and Target environments. By identifying entities (i.e., types and attributes, relationships for each dimension, and their specialization), novel foundations will be proposed to accumulate, query, and recommend currently dispersed orchestration knowledge in a structured framework.

9.5. Runtime Environment

Table IV maps the selected orchestration techniques onto the *Runtime Environment* dimensions as described in Section 7. We summarize our findings as follows:

—Nine of 11 approaches adopt a *centralized* execution model. This design choice is likely due to the simplicity of implementation. In comparison, *decentralized* orchestration requires an implementation that carefully considers discovery, synchronization, coordination, and security aspects of agents.
—Surprisingly, the value of *federated* cloud resources is largely underestimated. Most cloud resource orchestration techniques focus either on *private* or *public* cloud environments as their target environment, whereas only 1 of the 11 approaches we studied provide support for *federated* cloud resources management.
—The preference of a virtualization technique varies largely based on the types of resources (i.e., *Infrastructure*, *Platform*, or *Software*). All of the infrastructure-focused approaches that we analyzed adopt OS-level hypervisors as their virtualization technique. Other approaches that support *Platform* and *Software* resource adopt environment-level container managers.

Furthermore, we identify the following future directions in the evolution of *Runtime Environments*:

—*Runtime intelligence for autonomic and declarative orchestration.* Autonomic orchestration will play a key role in addressing crucial gaps in cloud computing [Toosi et al.

2014], as well as significantly improve overall productivity. Most existing work only apply orchestration strategies for specific aspects in isolation of each other, such as configuration [Xu et al. 2012], deployment [Antonescu et al. 2013; Beloglazov et al. 2012], and control [Schulte et al. 2015].

Accordingly, we believe that orchestration frameworks should be endowed with an embedded level of intelligence within their runtime environment, as well as the ability to manage themselves in accordance with high-level policies that are specified by users or administrators. For example, currently significant shortcomings exist to seamlessly integrate orchestration languages and techniques with scalable data processing platforms. In fact, such data platforms are essential for monitoring and enforcing SLAs, which involve capturing and analyzing large amounts of real-time data in big data analytics platforms (e.g., Hadoop and MapReduce). We believe that the *orchestration layer* should contain the intelligence responsible for specifying resource orchestration, while the *data processing layer* should contain "the intelligence responsible for data-flow and processing" [Lemos et al. 2016].

This could be achieved by more dynamic and knowledge-driven techniques that provide high-level reasoning about environment properties and automated support for policies provisioning to support a range of autonomic orchestration tasks, such as self-configuration and self-optimization, as well as self-healing and self-protecting tasks. DevOps will thus be able to describe resource requirements and constraints using declarative and orchestration-aware abstractions such as *State machines* (refer to Section 9.3). Orchestration runtimes may thus automatically translate such abstractions into efficient and technique-aware execution scripts.

—*Cloud service event summaries.* The ability for cloud orchestration platforms to gain the requisite intelligence about consumption patterns of deployed resources ensures compliance with cost and SLA constraints and improves resource orchestration processes in general (e.g., continuously fine-tuning defined policies in dynamic and evolving environments).

We therefore believe future work should develop concepts and techniques to model and capture event patterns and abstract them into meaningful concepts (e.g., characterizing states of an application or a service, state of a specific application component, behavior of users from a specific geolocation) that are suitable for cloud elastic resource orchestration purposes. Accordingly, we believe high-level language constructs to abstract and aggregate temporal and resource-relevant events over federated cloud services at various granularities will provide the key. These can be used to describe event summaries of knowledge about variations in resource requirements in terms of both aggregated resource consumption metrics (e.g., the number of API calls per second) and semantically meaningful event categories (e.g., moderate application load). Event summaries can be defined at various abstraction levels as a hierarchy to cater for context-based, fine- or coarse-grained analysis of resource requirements and consumption trends. Lower-level event summaries may be concrete (e.g., providing knowledge relevant to a fine-grained analysis of patterns for some specific cloud service such as Amazon DynamoDB). Higher-level event summaries may capture knowledge required for coarse-grained analysis of patterns relevant to a collection of resources (e.g., cluster, whole application).

10. CONCLUSION

Cloud resources and orchestration techniques are an effective technology, endowed with immense power to transform traditional *infrastructure*, *platform*, and *software* resources into elastic, measurable, on-demand self-service-based virtual components. In this extensive survey, we have studied a diverse mix of cloud resource orchestration techniques that include languages, services, standards, and tools. We presented a

novel taxonomy over a broad range of relevant dimensions, which we have applied to characterize and analyse various orchestration techniques. We contribute a systematic analysis of the most representative cloud resource orchestration techniques by evaluating and classifying them against the presented taxonomy. Towards the end of this contribution, we derive key open research issues based on the apparent technical gaps that were identified during the analysis. Accordingly, we propose a range of future directions as fruitful guidelines for the next generation of cloud orchestration.

ACKNOWLEDGMENTS

We thank Professor Frank Leymann (University of Stuttgart) and Dr. Brahim Medjahed (University of Michigan-Dearborn) for their initial comments.

REFERENCES

Brian Adler. 2011. Building Scalable Applications In the Cloud: Reference Architecture & Best Practices, RightScale Inc. Retrieved from https://s3.amazonaws.com/aws001/guided_trek/RightScale_White_Paper_Building_Scalable_Applications.pdf.

Jeannie Albrecht and et al. 2011. Distributed application configuration, management, and visualization with plush. *ACM Trans. Internet Technol.* 11, 2 (2011), 6.

Khalid Alhamazani, Rajiv Ranjan, Karan Mitra, Fethi Rabhi, Prem Prakash Jayaraman, Samee Ullah Khan, Adnene Guabtni, and Vasudha Bhatnagar. 2015. An overview of the commercial cloud monitoring tools: Research dimensions, design issues, and state-of-the-art. *Computing* 97, 4 (2015), 357–377.

AWS Amazon. 2011. AWS Cloud Formation. Retrieved from http://aws.amazon.com/cloudformation/.

AWS Amazon. 2015a. Amazon Relational Database Service—API Docuumentation. Retrieved from http://docs.aws.amazon.com/AmazonRDS/latest/APIReference/Welcome.html.

AWS Amazon. 2015b. AWS OpsWorks Template Snippets. Retrieved June 24, 2015 from http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/quickref-opsworks.html.

AWS Amazon. 2015c. EC2 Instances. Retrieved from http://aws.amazon.com/ec2/instance-types/.

Inc. Ansible. 2015. Ansible: Cloud Modules. Retrieved June 10, 2015 from http://docs.ansible.com/list_of_cloud_modules.html.

Alexandru-Florian Antonescu, Alvaro Gomes, Peter Robinson, and Torsten Braun. 2013. SLA-driven predictive orchestration for distributed cloud-based mobile services. In *Proceedings of the 2013 IEEE International Conference on Communications Workshops (ICC'13)*. IEEE, 738–743.

CA AppLogic. 2015. CA AppLogic Cloud Platform. Retrieved May 28, 2015 from http://www.ca.com/us/products/detail/ca-applogic.aspx.

Claudio A. Ardagna, Rasool Asal, Ernesto Damiani, and Quang Hieu Vu. 2015. From security to assurance in the cloud: A survey. *ACM Comput. Surv.* 48, 1, (July 2015) Article 2, 50 pages.

D. Ardagna et al. 2012. MODAClouds: A model-driven approach for the design and execution of applications on multiple Clouds. In *Proceedings of the 2012 ICSE Workshop on MISE*. 50–56.

Michael Armbrust et al. 2010. A view of cloud computing. *Commun. ACM* 53, 4 (April 2010), 50–58.

Amazon Auto Scaling. 2015. Auto Scaling for AWS cloud resources. Retrieved May 7, 2015 from http://aws.amazon.com/autoscaling/.

AWS. 2013a. Available commands for EC2 in AWS CLI. Retrieved May 7, 2015 from http://docs.aws.amazon.com/cli/latest/reference/ec2/index.html.

AWS. 2013b. AWS CLI. Retrieved from http://docs.aws.amazon.com/cli/latest/index.html.

AWS. 2015a. AWS SDK for Java. Retrieved November 10, 2015 from https://aws.amazon.com/sdk-for-java/.

AWS. 2015b. REST API for AWS S3. Retrieved November 10, 2015 from http://docs.aws.amazon.com/AmazonS3/latest/API/APIRest.html.

Apache CloudStack. 2016. Apache cloudstack: Open source cloud computing. Retrieved from Retrieved January 10, 2016 from https://cloudstack.apache.org/.

AWS CloudTrail. 2014. Security at scale: Logging in AWS. (2014).

Amazon CloudWatch. 2013. Monitoring for AWS cloud resources. Retrieved May 7, 2015 from http://aws.amazon.com/cloudwatch/.

Amazon Marketplace. 2012. Marketplace for AWS cloud resources. Retrieved May 7, 2015 from https://aws.amazon.com/marketplace.

Amazon Web Services. 2015a. Amazon EC2. Retrieved from http://aws.amazon.com/ec2/.

Amazon Web Services. 2015b. Amazon EC2 Container Registry. Retrieved from https://aws.amazon.com/ecr/.

Amazon Web Services. 2015c. AWS Management Console. Retrieved from https://aws.amazon.com/console/.

Arshdeep Bahga and Vijay K. Madisetti. 2013. Rapid prototyping of multitier cloud-based services and systems. *Computer* 46, 11 (2013), 76–83.

Wolfgang Barth. 2008. *Nagios: System and Network Monitoring*. No Starch Press.

Moshe Chai Barukh and Boualem Benatallah. 2013a. ServiceBase: A programming knowledge-base for service oriented development. In *Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA'13)*. Springer, 123–138.

Moshe Chai Barukh and Boualem Benatallah. 2013b. A toolkit for simplified web-services programming. In *Web Information Systems Engineering–WISE 2013*. Springer, 515–518.

Moshe Chai Barukh and Boualem Benatallah. 2014. ProcessBase: A hybrid process management platform. In *Proceedings of the International Conference on Service-Oriented Computing*. Springer, 16–31.

Erick Bauman, Gbadebo Ayoade, and Zhiqiang Lin. 2015. A survey on hypervisor-based monitoring: Approaches, applications, and evolutions. *ACM Comput. Surv.* 48, 1 (Aug. 2015), Article 10, 33 pages. DOI: http://dx.doi.org/10.1145/2775111

Anton Beloglazov, Jemal Abawajy, and Rajkumar Buyya. 2012. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future Gener. Comput. Syst.* 28, 5 (2012), 755–768.

Anton Beloglazov, Rajkumar Buyya, Young Choon Lee, Albert Zomaya, and others. 2011. A taxonomy and survey of energy-efficient data centers and cloud computing systems. *Adv. Comput.* 82, 2 (2011), 47–111.

Alexander Bergmayr, Alessandro Rossini, Nicolas Ferry, Geir Horn, Leire Orue-Echevarria, Arnor Solberg, and Manuel Wimmer. 2015. The evolution of cloudml and its applications. In *Proceedings of the 3rd International Workshop on Model-Driven Engineering on and for the Cloud 18th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2015)*. 13–18. Retrieved from http://ceur-ws.org/Vol-1563/paper3.pdf.

Jan A. Bergstra and Mark Burgess. 2008. A static theory of promises. *CoRR* abs/0810.3294 (2008). Retrieved from http://arxiv.org/abs/0810.3294.

Jan A. Bergstra and Mark Burgess. 2014. Promises, impositions, and other directionals. *arXiv Preprint arXiv:1401.3381* (2014).

Tobias Binz et al. 2013. OpenTOSCA–a runtime for TOSCA-based cloud applications. In *Service-Oriented Computing*. Springer, 692–695.

Bitnami. 2015. Bitnami makes it easy to run your favorite server apps anywhere. Retrieved May 28, 2015 from https://bitnami.com/learn_more.

Thomas J. Bittman. 2011. The Road Map From Virtualization to Cloud Computing. Retrieved March 2011 from https://www.gartner.com/doc/1572031.

Mark Burgess. 2007. Promise you a rose garden. Retrieved from http://markburgess.org/rosegarden.pdf.

Mark Burgess. 2009. Knowledge management and promises. In *Scalability of Networks and Services*. Springer, 95–107.

Mark Burgess. 2011. Testable system administration. *Commun. ACM* 54, 3 (2011), 44–49.

Mark Burgess and Oslo College. 1995. Cfengine: A site configuration engine. In *Proceedings of the USENIX Computing Systems, Vol*.

Mark Burgess and Alva L. Couch. 2006. Modeling next generation configuration management tools. In *Proceedings of the 20th Conference on Large Installation System Administration (LISA'06)*. 131–147.

Damon Cali. 2013. Introducing rumm: a Command Line Tool for the Rackspace Cloud. Retrieved June 9, 2015 from https://developer.rackspace.com/blog/introducing-rumm-a-command-line-tool-for-the-rackspace-cloud/.

Canonical. 2015. Juju Charm Store. Retrieved from https://jujucharms.com/store.

CenturyLink. 2015. Panamax: Docker Management for Humans. Retrieved from http://panamax.io/.

Clovis Chapman, Wolfgang Emmerich, Fermín Galán Márquez, Stuart Clayman, and Alex Galis. 2012. Software architecture definition for on-demand cloud provisioning. *Cluster Comput.* 15, 2 (2012), 79–100.

Muhammad Aufeef Chauhan, Muhammad Ali Babar, and Boualem Benatallah. 2016. Architecting cloud-enabled systems: A systematic survey of challenges and solutions. *Software: Practice and Experience* (2016).

Chef. 2015. About Recipes. Retrieved from https://docs.chef.io/recipes.html.

Peter Pin-Shan Chen. 1976. The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst.* 1, 1 (Mar. 1976), 9–36. DOI: http://dx.doi.org/10.1145/320434.320440

Shang-Wen Cheng and David Garlan. 2012. Stitch: A language for architecture-based self-adaptation. *J. Syst. Softw.* 85, 12 (Dec. 2012), 2860–2875. DOI: http://dx.doi.org/10.1016/j.jss.2012.02.060

Trieu C. Chieu et al. 2010. Solution-based deployment of complex application services on a cloud. In *IEEE International Conference on SOLI, 2010*. IEEE, 282–287.

Mark Chignell, James Cordy, Joanna Ng, and Yelena Yesha. 2010. *The Smart Internet: Current Research and Future Applications*. Vol. 6400. Springer Science & Business Media.

Cisco-Systems-Inc. 2011. Cloud: what an enterprise must know. Retrieved from http://www.cisco.com/en/US/solutions/collateral/ns340/ns517/ns224/ns836/ns976/white_paper_c11-617239.pdf.

Cloud-Foundry. 2016. The industry standard platform for cloud applications. Retrieved June 5, 2016 from https://www.cloudfoundry.org/.

Inc. CloudBees. 2016. CloudBees: The Enterprise Jenkins Company. Retrieved January 10, 2016 from https://www.cloudbees.com/.

Cloudlabs-Inc. 2015. Public Snaps. Retrieved from https://www.terminal.com/explore.

Alva L. Couch, John Hart, Elizabeth G. Idhaw, and Dominic Kallas. 2003. Seeking closure in an open world: A behavioral agent approach to configuration management. In *LISA*, Vol. 3. 125–148.

S. Crosby et al. 2009. Open virtualization format specification. *Standards and Technology, no. DSP0243 in DMTF Specifications, Distributed Management Task Force* (2009).

CS50. 2015. CS50 Appliance 19. Retrieved from https://manual.cs50.net/appliance/19/.

CSA. 2011. Security guidance for critical areas of focus in cloud computing. Retrieved November 2011 from https://cloudsecurityalliance.org/research/securityguidance/.

Yong Cui, Vojislav B. Misic, Rajkumar Buyya, and Dejan Milojicic. 2013. Guest editors' introduction: Special issue on cloud computing. *IEEE Trans. Parallel Distrib. Syst.* 24, 6 (2013).

Michael Cusumano. 2010. Cloud computing and saas as new computing platforms. *Commun. ACM* 53, 4 (2010), 27–29.

Cohesive Networks. 2016. Cohesive Networks: Home. Retrieved from https://cohesive.net/.

CA Technologies. 2013. INSSLR2 - Redundant HTTP Input Gateway with SSL Support. Retrieved July 10, 2015 from https://support.ca.com/cadocs/0/CA%20AppLogic%203%208-ENU/Bookshelf_Files/HTML/AppLogicDoc/index.htm?toc.htm?CatGatewayINSSLR2.html.

Clemens Danninger. 2015. Using constraint solvers to find valid software configurations. Retrieved from http://www.complang.tuwien.ac.at/raab/constraint_solvers.pdf.

James Darvell. 2016. Unikernels, Docker, and Why You Should Care. Retrieved November 25, 2016 from http://www.linuxjournal.com/content/unikernels-docker-and-why-you-should-care/.

Thomas Delaet, Wouter Joosen, and Bart Vanbrabant. 2010. A survey of system configuration tools. In *Proceedings of the 24th International Conference on LISA*. USENIX Association, 1–8. Retrieved from http://dl.acm.org/citation.cfm?id=1924976.1924977.

Zuohua Ding, Yuan Zhou, and MengChu Zhou. 2014. Modeling self-adaptive software systems with learning petri nets. In *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 464–467.

Nectar Directorate. 2016. Nectar: Australia's fastest growing researcher network. Retrieved January 10, 2016 from https://nectar.org.au/.

DMTF. 2010. Architecture for managing clouds — A white paper from the open cloud standards incubator. Retrieved June 2010 from http://dmtf.org/standards/cloud/.

Docker. 2015a. Docker Hub Registry. Retrieved from https://registry.hub.docker.com/.

Docker. 2015b. Overview of Docker Compose. Retrieved from https://docs.docker.com/compose/.

dotCloud. 2015. Online article. Retreived from https://www.dotcloud.com/dev-center/platform-documentation.

Robert Dukaric and Matjaz B. Juric. 2013. Towards a unified taxonomy and architecture of cloud frameworks. *Future Gener. Comput. Syst.* 29, 5 (2013), 1196–1210.

Marlon Dumas, Marcello La Rosa, Jan Mendling, and Hajo A Reijers. 2013. *Fundamentals of Business Process Management*. Springer.

Erik Elmroth and Lars Larsson. 2009. Interfaces for placement, migration, and monitoring of virtual machines in federated clouds. In *Proceedings of the 2009 8th International Conference on Grid and Cooperative Computing (GCC'09)*. IEEE, 253–260.

Finally.io. 2014. finally.io. Retrieved February 8, 2015 from https://www.finally.io/.

Inc. Engine Yard. 2016. Engine Yard. Retrieved January 10, 2016 from https://www.engineyard.com/.

Daren Fang, Xiaodong Liu, Imed Romdhani, and Claus Pahl. 2015. An approach to unified cloud service access, manipulation and dynamic orchestration via semantic cloud service operation specification framework. *J. Cloud Comput.* 4, 1 (2015), 1.

Wei Fang, ZhiHui Lu, Jie Wu, and ZhenYin Cao. 2012. RPPS: A novel resource prediction and provisioning scheme in cloud data center. In *Proceedings of the 2012 IEEE 9th International Conference on Services Computing (SCC)*. IEEE, 609–616.

Kaniz Fatema, Vincent C. Emeakaroha, Philip D. Healy, John P. Morrison, and Theo Lynn. 2014. A survey of Cloud monitoring tools: Taxonomy, capabilities and objectives. *J. Parallel Distrib. Comput.* 74, 10 (2014), 2918–2933.

Joerg Fritsch. 2015. Security properties of Containers managed by Docker. Retrieved June 5, 2015 from https://www.gartner.com/doc/2956826/security-properties-containers-managed-docker.

Gartner. 2013. Gartner Says Cloud Computing Will Become the Bulk of New IT Spend by 2016. Retrieved November 24, 2015 from http://www.gartner.com/newsroom/id/2613015.

Inc. Gartner. 2014. Gartner Survey Reveals That SaaS Deployments Are Now Mission Critical. Retrieved July 14, 2015 from http://www.gartner.com/newsroom/id/2923217.

Wolfgang Gerlach et al. 2014. Skyport: Container-based execution environment management for multi-cloud scientific workflows. In *Proceedings of the 5th International Workshop on Data-Intensive Computing in the Clouds*. IEEE Press, 25–32.

Patrick Goldsack et al. 2009. The smartfrog configuration management framework. *ACM SIGOPS Operat. Syst. Rev.* 43, 1 (2009), 16–25.

Patrick Goldsack, Julio Guijarro, Steve Loughran, Alistair N. Coles, Andrew Farrell, Antonio Lain, Paul Murray, and Peter Toft. 2009. The smartfrog configuration management framework. *Operat. Syst. Rev.* 43, 1 (2009), 16–25.

Google. 2015a. Container Registry: Fast, private Docker image storage on Google Cloud Platform. Retrieved November 17, 2015 from https://cloud.google.com/container-registry/.

Google. 2015b. Google App Engine: Platform as a Service. Retrieved June 8, 2015 from https://cloud.google.com/appengine/docs.

Christophe Gravier, Julien Subercaze, Amro Najjar, Frederique Laforest, Xavier Serpaggi, and Olivier Boissier. 2015. Context awareness as a service for cloud resource optimization. *IEEE Internet Comput.* 19, 1 (2015), 28–34.

Nikolay Grozev and Rajkumar Buyya. 2014. Inter-cloud architectures and application brokering: Taxonomy and survey. *Softw.: Pract. Exper.* 44, 3 (2014), 369–390.

Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. 2006. Enforcing performance isolation across virtual machines in xen. In *Proceedings of the ACM International Conference on Middleware (Middleware'06)*. Springer-Verlag, New York, NY, 342–362.

Mohammad Hajjat, Xin Sun, Yu-Wei Eric Sung, David Maltz, Sanjay Rao, Kunwadee Sripanidkulchai, and Mohit Tawarmalani. 2011. Cloudward bound: Planning for beneficial migration of enterprise applications to the cloud. *ACM SIGCOMM Comput. Commun. Rev.* 41, 4 (2011), 243–254.

Abdul Hameed, Alireza Khoshkbarforoushha, Rajiv Ranjan, Prem Prakash Jayaraman, Joanna Kolodziej, Pavan Balaji, Sherali Zeadally, Qutaibah Marwan Malluhi, Nikos Tziritas, Abhinav Vishnu, et al. 2016. A survey and taxonomy on energy efficient resource allocation techniques for cloud computing systems. *Computing* 98, 7 (2016), 751–774.

Ahmad Fadzil M. Hani, Irving Vitra Paputungan, and Mohd Fadzil Hassan. 2015. Renegotiation in service level agreement management for a cloud-based system. *Comput. Surv.* 47, 3 (2015), 51.

Mitchell Hashimoto. 2013. *Vagrant: Up and Running*. O'Reilly Media, Inc.

Christina N. Hoefer, Georgios Karagiannis, and et al. 2010. Taxonomy of cloud computing services. In *Proceedings of the 2010 IEEE Globecom Workshops*. IEEE, 1345–1350.

Ben Hosmer. 2012. Getting started with salt stack–the other configuration management system built with python. *Linux J.* 2012, 223 (2012), 3.

Wei Huang et al. 2015. The state of public infrastructure-as-a-service cloud security. *ACM Comput. Surv.* 47, 4 (June 2015), Article 68, 31 pages. DOI:http://dx.doi.org/10.1145/2767181

Hewlett Packard Enterprise Development. 2016. HPE Helion Eucalyptus: Open source hybrid cloud software for AWS users.Retrieved January 10, 2016 from http://www8.hp.com/us/en/cloud/helion-eucalyptus-overview.html.

Intel Corporation. 2015. Cloud computing taxonomy and ecosystem analysis. Retrieved September 2012 from http://www.intel.com/content/dam/doc/case-study/intel-it-cloudcomputing-taxonomy-ecosystem-analysis-study.pdf.

Waheed Iqbal, Matthew N. Dailey, David Carrera, and Paul Janecek. 2011. Adaptive resource provisioning for read intensive multi-tier applications in the cloud. *Future Gener. Comput. Syst.* 27, 6 (2011), 871–879.

Sadeka Islam, Jacky Keung, Kevin Lee, and Anna Liu. 2012. Empirical prediction models for adaptive resource provisioning in the cloud. *Future Gener. Comput. Syst.* 28, 1 (2012), 155–162.

Brendan Jennings and Rolf Stadler. 2014. Resource management in clouds: Survey and research challenges. *J. Netw. Syst. Manag.* (2014), 1–53.

Yexi Jiang, Chang-shing Perng, Tao Li, and Rong Chang. 2011. Asap: A self-adaptive prediction system for instant cloud resource demand provisioning. In *Proceedings of the 2011 IEEE 11th International Conference on Data Mining (ICDM'11)*. IEEE, 1104–1109.

Matjaz B. Juric and Denis Weerasiri. 2014. *WS-BPEL 2.0 Beginner's Guide*. Packt Publishing Ltd.

Eleni Kamateri, Nikolaos Loutas, Dimitris Zeginis, James Ahtes, Francesco D'Andria, Stefano Bocconi, Panagiotis Gouvas, Giannis Ledakis, Franco Ravagli, Oleksandr Lobunets, and others. 2013. Cloud4soa: A semantic-interoperability paas solution for multi-cloud platform management and portability. In *Proceedings of the European Conference on Service-Oriented and Cloud Computing*. Springer, 64–78.

Luke Kanies. 2006. Puppet: Next-generation configuration management. *USENIX Mag.* 31, 1 (2006), 19–25.

B. Khasnabish, J. Chu, S. Ma, Y. Meng, N. So, P. Unbehagen, et al. 2011. IEFT cloud reference framework. Retrieved from http://tools.ietf.org/html/draft-khasnabishcloud-reference-framework-02.

Alireza Khoshkbarforoushha, Meisong Wang, Rajiv Ranjan, Lizhe Wang, Leila Alem, Samee U. Khan, and Boualem Benatallah. 2016. Dimensions for evaluating cloud resource orchestration frameworks. *Computer* 49, 2 (2016), 24–33.

Hyunjoo Kim and Manish Parashar. 2011. CometCloud: An autonomic cloud engine. *Cloud Computing: Principles and Paradigms* (2011), 275–297.

Johannes Kirschnick et al. 2012. Towards an architecture for deploying elastic services in the cloud. *Softw. Pract. Exper.* 42, 4 (Apr. 2012), 395–408. DOI:http://dx.doi.org/10.1002/spe.1090

Alexander V. Konstantinou et al. 2009. An architecture for virtual solution composition and deployment in infrastructure clouds. In *Proceedings of the 3rd International Workshop on VTDC*. ACM, 9–18.

Oliver Kopp et al. 2012. BPMN4TOSCA: A domain-specific language to model management plans for composite applications. In *Business Process Model and Notation*. Springer, 38–52.

Peter Laird. 2008. Cloud Taxonomy. Retrieved September 2008 from https://sites.google.com/site/saaslink/Laird_CloudMap_Sept2008.png.

C. Larman and V. R. Basili. 2003. Iterative and incremental developments. A brief history. *Computer* 36, 6 (June 2003), 47–56. DOI:http://dx.doi.org/10.1109/MC.2003.1204375

George Lawton. 2005. LAMP lights enterprise development efforts. *Computer* 38, 9 (2005), 0018–20.

Angel Lagares Lemos, Florian Daniel, and Boualem Benatallah. 2016. Web service composition: A survey of techniques and tools. *ACM Comput. Surv.* 48, 3 (2016), 33.

Grace Lewis et al. 2013. Role of standards in cloud-computing interoperability. In *Proceedings of the 2013 46th Hawaii International Conference on System Sciences (HICSS)*. IEEE, 1652–1661.

Christoph Fehling Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter. 2014. Cloud computing patterns. Springer, Wien. doi 10 (2014): 978–3.

LinuxContainers.org. 2015. What's LXC? Retrieved June 8, 2015 from https://linuxcontainers.org/lxc/introduction/.

Changbin Liu, Boon Thau Loo, and Yun Mao. 2011a. Declarative automated cloud resource orchestration. In *Proceedings of the SOCC'11*. ACM, Article 26, 8 pages.

Changbin Liu, Yun Mao, Jacobus Van der Merwe, and Mary Fernandez. 2011b. Cloud resource orchestration: A data-centric approach. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR'11)*. 1–8.

Fang Liu, Jin Tong, Jian Mao, Robert Bohn, John Messina, Lee Badger, and Dawn Leaf. 2011c. NIST cloud computing reference architecture. *NIST Spec. Publ.* 500, 2011 (2011), 292.

Scott Lowe. 2011. *Mastering VMware vSphere 5*. John Wiley & Sons.

Hongbin Lu, M. Shtern, B. Simmons, M. Smit, and M. Litoiu. 2013. Pattern-based deployment service for next generation clouds. In *Proceedings of the 2013 IEEE 9th World Congress on Services (SERVICES)*. 464–471.

Heiko Ludwig, Alexander Keller, Asit Dan, Richard King, and Richard Franck. 2003. A service level agreement language for dynamic electronic services. *Electron. Commerce Res.* 3, 1–2 (2003), 43–59.

Linux Foundation. 2015a. Open Container Initiative. Retrieved September 24, 2015 from https://www.opencontainers.org/. (2015).

Linux Foundation. 2015b. Open Container Project. Retrieved from http://www.opencontainers.org/.

MadeiraCloud. 2015. CloudFielder: Policy as a Service, for your cloud infrastrucutre. Retrieved October 10, 2015 from http://cloudfielder.com/.

David J. Malan. 2015. CS50. Retrieved June 8, 2015 from https://cs50.harvard.edu/.

Ebrahim H. Mamdani. 1974. Application of fuzzy algorithms for control of simple dynamic plant. In *Proceedings of the Institution of Electrical Engineers*, Vol. 121. IET, 1585–1588.

Zoltán Ádám Mann. 2015. Allocation of virtual machines in cloud data centers-a survey of problem models and optimization algorithms. *ACM Comput. Surv.* 48, 1 (Aug. 2015), Article 11, 34 pages.

Toni Mastelic, Ariel Oleksiak, Holger Claussen, Ivona Brandic, Jean-Marc Pierson, and Athanasios V. Vasilakos. 2014. Cloud computing: Survey on energy efficiency. *ACM Comput. Surv.* 47, 2 (Dec. 2014), Article 33, 36 pages. DOI: http://dx.doi.org/10.1145/2656204

Michael Menzel, Rajiv Ranjan, Lizhe Wang, Samee U. Khan, and Jinjun Chen. 2015. CloudGenius: A hybrid decision support method for automating the migration of web application clusters to public clouds. *IEEE Trans. Comput.* 64, 5 (2015), 1336–1348.

Thijs Metsch, Andy Edmonds, R. Nyrén, and A. Papaspyrou. 2010. Open cloud computing interface–core. In Open Grid Forum, OCCI-WG, Specification Document.

Brenda M. Michelson. 2006. Event-driven architecture overview. *Patricia Seybold Group* 2 (2006). Retrieved from http://elementallinks.com/el-reports/EventDrivenArchitectureOverview_ElementalLinks_Feb2011.pdf.

Neil Middleton, Richard Schneeman, and others. 2013. *Heroku: Up and Running*. O'Reilly Media, Inc.

M. Mishra, A. Das, P. Kulkarni, and A. Sahoo. 2012. Dynamic resource management using virtual machine migrations. *IEEE Commun. Mag.* 50, 9 (Sept. 2012), 34–40.

Madhurranjan Mohaan and Ramesh Raithatha. 2014. *Learning Ansible*. Packt Publishing Ltd.

Francesco Moscato, Rocco Aversa, Beniamino Di Martino, Teodor-Florin Fortiş, and Victor Munteanu. 2011. An analysis of mosaic ontology for cloud resources annotation. In *Proceedings of the 2011 Federated Conference on Computer Science and Information Systems (FedCSIS)*. IEEE, 973–980.

Nitrous. 2013. nitrous.io. Retrieved May 7, 2015 from https://nitrous.io.

OASIS 2013. *Topology and Orchestration Specification for Cloud Applications (TOSCA), Version 1.0*. OASIS.

Kiran Oliver. 2015. TNS Markers: The Comparison and Context of Unikernels and Containers. Retrieved November 25, 2016 from http://thenewstack.io/the-comparison-and-context-of-unikernels-and-containers/.

OMG 2011. *Business Process Model and Notation (BPMN), Version 2.0*. OMG.

OpenCrowd. 2010. Cloud Taxonomy. Retrieved from http://cloudtaxonomy.opencrowd.com.

OpenStack.org. 2015a. Open source software for creating private and public clouds. Retrieved May 30, 2015 from https://www.openstack.org/.

OpenStack.org. 2015b. OpenStack Orchestration. Retrieved from https://wiki.openstack.org/wiki/Heat.

Oracle Corporation. 2011. Oracle reference architecture—cloud infrastructure. Retrieved November 2011 from http://www.oracle.com/technetwork/topics/entarch/oracle-ra-cloudinfrastructure-r3-0-1395892.pdf.

Suraj Pandey, Linlin Wu, Siddeswara Mayura Guru, and Rajkumar Buyya. 2010. A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments. In *Proceedings of the 2010 24th IEEE International Conference on Advanced Information Networking and Applications (AINA'10)*. IEEE, 400–407.

Manish Parashar and Salim Hariri. 2005. Autonomic computing: An overview. In *Unconventional Programming Paradigms*. Springer, 257–269.

Dana Petcu. 2014. Consuming resources and services from multiple clouds. *J. Grid Comput.* 12, 2 (2014), 321–345.

Google Cloud Platform. 2015. Cloud SDK. Retrieved from https://cloud.google.com/sdk/.

OpenNebula Project. 2016. OpenNebula—Flexible Enterprise Cloud Made Simple. Retrieved January 10, 2016 from http://opennebula.org/.

Puppet. 2015. Overview of Orchestration Topics. Retrieved October 10, 2015 from https://docs.puppetlabs.com/pe/latest/orchestration_overview.html.

Puppet Labs. 2015a. Publishing Modules on the Puppet Forge. Retrieved June 8, 2015 from https://docs.puppetlabs.com/puppet/latest/reference/modules_publishing.html.

Puppet Labs. 2015b. Puppet Enterprise. Retrieved from https://puppetlabs.com/puppet/puppet-enterprise.

Puppet Labs. 2015c. Type Reference. Retrieved June 8, 2015 from https://docs.puppetlabs.com/references/latest/type.html.

Rackspace. 2015. Rackspace: API Documentation.Retrieved from http://docs.rackspace.com/.

Fahimeh Ramezani, Jie Lu, and Faheem Hussain. 2013. An online fuzzy decision support system for resource management in cloud environments. In *Proceedings of the 2013 Joint IFSA World Congress and NAFIPS Annual Meeting (IFSA/NAFIPS)*. IEEE, 754–759.

Rajiv Ranjan, Boualem Benatallah, Schahram Dustdar, and Michael P. Papazoglou. 2015. Cloud resource orchestration programming: Overview, issues, and directions. *IEEE Internet Comput.* 19, 5 (2015), 46–56.

Rajiv Ranjan, Rajkumar Buyya, and Surya Nepal. 2013. Editorial: Model-driven provisioning of application services in hybrid computing environments. *Future Gener. Comput. Syst.* 29, 5 (July 2013), 1211–1215. DOI:http://dx.doi.org/10.1016/j.future.2013.01.007

Real-Status-Ltd. 2015. A visibly different approach to cross-domain, hybrid IT management. Retrieved October 7, 2015 from http://www.hyperglance.com/wp-content/uploads/2015/08/HyperglanceDatasheet_Final 1.pdf.

Paul Resnick and Hal R. Varian. 1997. Recommender systems. *Commun. ACM* 40, 3 (1997), 56–58.

Stefan Ried, Holger Kisker, and Pascal Matzke. 2010. The evolution of cloud computing markets. *Forrester Res.* Retrieved from http://fm.sap.com/data/upload/files/forrester%20-%20the%20evolution %20of%20cloud%20computing%20markets.pdf.

RightScale. 2016. Self-Service. Retrieved from http://rightscale.com/products-and-services/products/self-service.

Bhaskar Prasad Rimal, Eunmi Choi, and Ian Lumb. 2009. A taxonomy and survey of cloud computing systems. In *2009 Fifth International Joint Conference on INC, IMS and IDC (2009)*. 44–51.

Rami Rosen. 2013. Resource management: Linux kernel namespaces and cgroups. Haifux. May (2013).

Todd Rosner. 2013. *Learning AWS OpsWorks*. Packt Publishing Ltd.

Arpan Roy, Santonu Sarkar, Rajeshwari Ganesan, and Geetika Goel. 2015. Secure the cloud: From the perspective of a service-oriented organization. *ACM Comput. Surv.* 47, 3 (2015), 41.

Navin Sabharwal. 2014. *Automation Through Chef Opscode*. APress.

H. Sato, A. Kanai, and S. Tanimoto. 2010. A cloud trust model in a security aware cloud. In *Proceedings of the 2010 10th IEEE/IPSJ International Symposium on Applications and the Internet (SAINT)*. 121–124.

Benjamin Satzger et al. 2013. Winds of change: From vendor lock-in to the meta cloud. *IEEE Internet Comput.* 17, 1 (2013), 69–73.

Pete Sawyer, Raul Mazo, Daniel Diaz, Camille Salinesi, and Danny Hughes. 2012. Using constraint programming to manage configurations in self-adaptive systems. *Computer* 10 (2012), 56–63.

Stefan Schulte, Christian Janiesch, Srikumar Venugopal, Ingo Weber, and Philipp Hoenisch. 2015. Elastic business process management: State of the art and open challenges for BPM in the cloud. *Future Gener. Comput. Syst.* 46 (2015), 36–50.

Shipyard. 2015. Shipyard Walkthrough. Retrieved from https://shipyard-project.com/walkthrough/.

Junaid Shuja, Kashif Bilal, Sajjad A. Madani, Mazliza Othman, Rajiv Ranjan, Pavan Balaji, and Samee U. Khan. 2014. Survey of techniques and architectures for designing energy-efficient data centers. *IEEE Systems Journal* 10, 2 (2016), 507–519.

Sukhpal Singh and Inderveer Chana. 2015. QoS-aware autonomic resource management in cloud computing: A systematic review. *ACM Comput. Surv.* 48, 3 (Dec. 2015), Article 42, 46 pages.

Sukhpal Singh and Inderveer Chana. 2016. QoS-aware autonomic resource management in cloud computing: A systematic review. *ACM Comput. Surv.* 48, 3 (2016), 42.

James Skene, Franco Raimondi, and Wolfgang Emmerich. 2010. Service-level agreements for electronic services. *IEEE Trans. Softw. Eng.* 36, 2 (2010), 288–304.

M. Smit, B. Simmons, M. Shtern, and M. Litoiu. 2013. Supporting application development with structured queries in the cloud. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. 1213–1216.

StackEngine. 2015. StackEngine Container Application Center. Retrieved from http://stackengine.com/product/.

The Apache Software Foundation. 2014a. An API that abstracts the differents between clouds. Retrieved June 10, 2015 from https://deltacloud.apache.org/.

The Apache Software Foundation. 2014b. Compute Guide. Retrieved November 10, 2015 from https://jclouds.apache.org/start/compute/.

The Apache Software Foundation. 2014c. The Java Multi-Cloud Toolkit. Retrieved June 10, 2015 from https://jclouds.apache.org/.

The Apache Software Foundation. 2015c. One Interface To Rule Them All. Retrieved June 10, 2015 from https://libcloud.apache.org/.

R. W. Thrash. 2010. Building a Cloud Computing Specification: Fundamental Engineering for Optimizing Cloud Computing Initiatives. Retrieved March 2010 from http://assets1.csc.com/innovation/downloads/CSC_Papers_2010_Building_a_Cloud_Computing_Specification.pdf.

Doug Tidwell. 2009. The Simple Cloud API: Writing portable, interoperable applications for the cloud. Retrieved from http://www.ibm.com/developerworks/library/os-simplecloud/.

Adel Nadjaran Toosi, Rodrigo N. Calheiros, and Rajkumar Buyya. 2014. Interconnected cloud computing environments: Challenges, taxonomy, and survey. *ACM Comput. Surv.* 47, 1 (2014), 7.

James Turnbull. 2014. *The Docker Book: Containerization Is the New Virtualization*. James Turnbull.

TIBCO Software Inc. 2014. *Event Processing with State Machines*. Technical Report.

Ubuntu. 2013. Juju. Retrieved from http://www.ubuntu.com/cloud/tools/juju.

Ubuntu Juju. 2015a. Charm Store Policy. Retrieved June 8, 2015 from https://juju.ubuntu.com/docs/authors-charm-policy.html.

Ubuntu Juju. 2015b. What is a relation? Retrieved June 8, 2015 from https://jujucharms.com/docs/stable/authors-interfaces.

Peter Van Roy et al. 2009. Programming paradigms for dummies: What every programmer should know. *New Comput. Paradigms Comput. Music* 104 (2009).

David Villegas et al. 2012. Cloud federation in a layered service model. *J. Comput. Syst. Sci.* 78, 5 (Sept. 2012), 1330–1344. DOI:http://dx.doi.org/10.1016/j.jcss.2011.12.017

VisualOps. 2015. VisualOps - WYSIWYG for your cloud. Retrieved from http://docs.visualops.io/.

Inc. VMware. 2015. Understanding virtual machine snapshots in VMware ESXi and ESX (1015180). Retrieved November 17, 2015 from http://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=1015180.

Lizhe Wang, Rajiv Ranjan, Jinjun Chen, and Boualem Benatallah. 2012. *Cloud Computing: Methodology, Systems, and Applications*. CRC Press.

Denis Weerasiri, Moshe Chai Barukh, Boualem Benatallah, and Cao Jian. 2016. CloudMap: A visual notation for representing and managing cloud resources. In *Proceedings of the International Conference on Advanced Information Systems Engineering*. Springer, 427–443.

Denis Weerasiri and Boualem Benatallah. 2015. Unified representation and reuse of federated cloud resources configuration knowledge. In *Proceedings of the 2015 IEEE 19th International Enterprise Distributed Object Computing Conference (EDOC)*. 142–150.

Denis Weerasiri, Boualem Benatallah, and Moshe Chai Barukh. 2015. Process-driven configuration of federated cloud resources. In *Database Systems for Advanced Applications*. Springer, 334–350.

Yi Wei and M. Brian Blake. 2013. Adaptive service workflow configuration and agent-based virtual resource management in the cloud*. In *Proceedings of the 2013 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 279–284.

Johannes Wettinger, Uwe Breitenbücher, and Frank Leymann. 2014. Standards-based devops automation and integration using TOSCA. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*. IEEE Computer Society, 59–68.

Johannes Wettinger et al. 2014. Unified invocation of scripts and services for provisioning, deployment, and management of cloud applications based on TOSCA. In *CLOSER 2014*. SciTePress, 559–568.

Matthew S. Wilson. 2009. Constructing and managing appliances for cloud deployments from repositories of reusable components. In *Proceedings of the 2009 Conference on HotCloud'09*. USENIX Association.

Erik Wittern, Alexander Lenk, Sebastian Bartenbach, and Tobias Braeuer. 2014. Feature-based configuration of vendor-independent deployments on iaas. In *Proceedings of the 2014 IEEE 18th International Enterprise Distributed Object Computing Conference (EDOC'14)*. IEEE, 128–135.

Cheng-Zhong Xu, Jia Rao, and Xiangping Bu. 2012. URL: A unified reinforcement learning approach for autonomic cloud management. *J. Parallel Distrib. Comput.* 72, 2 (2012), 95–105.

Zhen Ye, Sajib Mistry, Athman Bouguettaya, and Hai Dong. 2016. Long-term QoS-aware cloud service composition using multivariate time series analysis. *IEEE Trans. Serv. Comput.* 9, 3 (2016), 382–393.

Eric Yuan, Naeem Esfahani, and Sam Malek. 2014. A systematic survey of self-protecting software systems. *ACM Trans. Auton. Adapt. Syst.* 8, 4 (2014), 17.

Rostyslav Zabolotnyi, Philipp Leitner, and Schahram Dustdar. 2014. Profiling-based task scheduling for factory-worker applications in infrastructure-as-a-service clouds. In *Proceedings of the 2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA'14)*. IEEE, 119–126.

Rostyslav Zabolotnyi, Philipp Leitner, Stefan Schulte, and Schahram Dustdar. 2015. SPEEDL–A declarative event-based language to define the scaling behavior of cloud applications. In *Proceedings of the 2015 IEEE World Congress on Services (SERVICES'15)*. 71–78. DOI:http://dx.doi.org/10.1109/SERVICES.2015.19

Peter Zadrozny and Raghu Kodali. 2013. *Big Data Analytics Using Splunk: Deriving Operational Intelligence from Social Media, Machine Data, Existing Data Warehouses, and Other Real-Time Streaming Sources*.

Liangzhao Zeng, B. Benatallah, A. H. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. 2004. QoS-aware middleware for web services composition. *IEEE Trans. Softw. Eng.* 30, 5 (May 2004), 311–327. DOI:http://dx.doi.org/10.1109/TSE.2004.11

Zhi-Hui Zhan, Xiao-Fang Liu, Yue-Jiao Gong, Jun Zhang, Henry Shu-Hung Chung, and Yun Li. 2015a. Cloud computing resource scheduling and a survey of its evolutionary approaches. *ACM Comput. Surv.* 47, 4, Article 63 (July 2015), 33 pages. DOI:http://dx.doi.org/10.1145/2788397

Zhi-Hui Zhan, Xiao-Fang Liu, Yue-Jiao Gong, Jun Zhang, Henry Shu-Hung Chung, and Yun Li. 2015b. Cloud computing resource scheduling and a survey of its evolutionary approaches. *ACM Comput. Surv.* 47, 4 (2015), 63.

Miranda Zhang, Rajiv Ranjan, Armin Haller, Dimitrios Georgakopoulos, Michael Menzel, and Surya Nepal. 2012b. An ontology-based system for Cloud infrastructure services' discovery. In *Proceedings of the International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom'12)*. IEEE.

Miranda Zhang, Rajiv Ranjan, Anna Haller, Dimitrios Georgakopoulos, and Peter Strazdins. 2012a. Investigating decision support techniques for automating cloud service selection. In *Proceedings of the 2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom'12)*. IEEE, 759–764.

Miranda Zhang, Rajiv Ranjan, Surya Nepal, Michael Menzel, and Armin Haller. 2012c. A declarative recommender system for cloud infrastructure services selection. In *Proceedings of the 9th International Conference on Economics of Grids, Clouds, Systems, and Services (GECON'12)*. Springer-Verlag, Berlin, 102–113. DOI:http://dx.doi.org/10.1007/978-3-642-35194-5_8

Xinwen Zhang, Anugeetha Kunjithapatham, Sangoh Jeong, and Simon Gibbs. 2011. Towards an elastic application model for augmenting the computing capabilities of mobile devices with cloud computing. *Mobile Netw. Appl.* 16, 3 (2011), 270–284.

Received August 2016; revised December 2016; accepted January 2017