Production Deployment Tools for IaaSes: an Overall Model and Survey

Hélène Coullon, Dimitri Pertin, Christian Pérez

Production Deployment Tools for IaaSes: an Overall Model and Survey

Hélène Coullon, Dimitri Pertin
Inria, LS2N, IMT Atlantique
Nantes, France
Email: helene.coullon@inria.fr, dimitri.pertin@inria.fr

Christian Perez
Inria, LIP, ENS Lyon
Lyon, France
Email: christian.perez@inria.fr

***Abstract*—Emerging applications for the Internet of Things (IoT) are complex programs which are composed of multiple modules (or services). For scalability, reliability and performance, modular applications are distributed on infrastructures that support utility computing (*e.g.*, Cloud, Fog). In order to simply operate such infrastructures, an Infrastructure-as-a-Service (IaaS) manager is required. OpenStack is the de-facto open-source solution to address the IaaS level of the Cloud paradigm. However, OpenStack is itself a large modular application composed of more than 150 modules that make it hard to deploy manually.**

To fully understand how IaaSes are deployed today, we propose in this paper an overall model of the application deployment process which describes each step with their interactions. This model then serves as the basis to analyse five different deployment tools used to deploy OpenStack in production: Kolla, Enos, Juju, Kubernetes, and TripleO. Finally, a comparison is provided and the results are discussed to extend this analysis.

I. Introduction

Emerging applications for the Internet of Things (IoT) are composed of multiple modules (or services) that interact together through well defined interfaces [1]. For scalability, reliability and performance, modular applications are distributed on infrastructures that support utility computing (*e.g.*, Cloud, Fog). In order to simply operate compute, network and storage resources of such infrastructures, an Infrastructure-as-a-Service (IaaS) manager is required. However, IaaS managers are themself complex modular applications that are hard to deploy manually on such dynamic and global network infrastructures [2], [3]. Thus, we investigate in this paper how IaaSes are deployed today.

OpenStack [4] is the de-facto open-source solution to address the IaaS level of the Cloud paradigm. Its community has gathered more than 500 organizations (*e.g.*, Google, IBM, Intel), that have produced more than 20 million lines of code in six years. While its adoption is still growing in various domains such as public administrations, e-commerce and science,[1] OpenStack is itself a large modular distributed application composed of more than 30 projects that manage the different aspects of an IaaS (*e.g.*, compute instances, storage, network). While they are not all mandatory to deploy an operable IaaS, these projects gather more than 150 services whose deployment is a tedious challenge that cannot be handled manually on dynamic and large-scale infrastructures [5].

For this reason, this paper investigates how a complex distributed software like OpenStack is deployed today. More specifically, the contributions of this paper are: (i) the description of a deployment process depicted in Section II; (ii) its use in Section III to compare five existing deployment tools already used to deploy OpenStack, *i.e.*, Kolla, Enos, Juju, Kubernetes and TripleO; and (iii) a discussion in Section IV that extends the previous analysis. Finally, the paper gives a state of the art on related surveys in Section V and a conclusion that opens to research perspectives in Section VI.

[1]See http://superuser.openstack.org/ for further information

II. Deployment model

This section first introduces a set of definitions used throughout this paper. Then, we present a generic deployment model used as a basis to study and compare different deployment tools in Section III.

A. Definitions

Distributed Software: A distributed software, or a software, defines the overall entity to deploy on a distributed infrastructure. It could be an end-user application, a middleware, a framework, or a distributed system, such as OpenStack which is studied in this paper.

Service: The distributed software is composed of different services, also called modules in this paper. Thus, services are subparts of the overall software to distribute (configure and install) on the infrastructure.

Resource: A resource defines an entity that will host one or multiple services of the overall software. A resource could be a bare-metal machine (*i.e.*, a physical server without any operating system) or a virtualized system (either container of virtual machine).

B. Model

Figure 1 graphically presents an overview of a generic deployment process that will be used in the next section to study deployment tools. This model is composed of rounded rectangles that we call components or steps. In this model, each step is usually found in existing deployment tools. The arrow semantic follows the usual workflow or dataflow rules: the execution of the step at the destination of an arrow is started when the source step is finished; output data is produced by the source step and is transferred to the destination step.

As it will be shown in Section III, depending on the studied tool, this deployment model can be adapted (*e.g.*, order of steps, merged steps).

a) Service packaging: The first step prepares the software to be deployed. The service packaging aims at splitting a software into different services, and at adapting these services to the chosen deployment process. For example, a deployment process designed for containers requires to encapsulate the service with its dependencies in a container image.

b) Template: Once the services are packaged, a *Template* can be written by the operator (*i.e.*, the user of the deployment tool). A template is composed of two parts. First, a description of the deployment topology that defines the services (*e.g.*, a database reachable via a port), resources (*e.g.*, a virtual machine running Debian), their associated constraints (*e.g.*, CPU, RAM, disk capacities) and the relations between them (*e.g.*, set the database service on a virtual machine). Secondly, information about the orchestration of services (*e.g.*, scaling rules) can be set to manage their lifecycle. Since this step is always managed by an operator in existing tools, it is represented with a human icon in Figure 1.

c) Dependencies: This step takes into account the relationships declared in the template to build a dependency graph or a sequential order of deployment. This graph could be needed by any of the following steps, as depicted in the model by the dashed rectangle.

d) Placement: The *Placement* consists in solving complex multi-objectives bin-packing problems, or scheduling problem. The problem is to fit n software modules or virtual machines to m available physical resources while respecting their constraints, and while answering objective functions (*e.g.*, minimizing the number of allocated machines). As this problem is NP-hard, solvers can use exact algorithms for small size cases only [6], otherwise heuristics are needed to scale up [7]. As depicted on the figure, this step requires information on the available resources and their current configuration.

e) Allocation: The *Allocation* step starts when the Placement is over. It consists in reserving resources chosen in the Placement step.

f) Provisioning: The *Provisioning* step defines the action of preparing the resources for their future role, *i.e.*, hosting software services. In this paper, the provisioning step manages the operating system installation (if needed) as well as any installation requirement of the deployment tool (for example Docker when containers are used for deployment). Thus, *Provisioning* is more or less heavy, depending on the kind of available resources and on the kind of deployment process (*i.e.*, install an operating system takes longer than install a container system).

g) Configuration: Once the resources are provisioned, it is necessary to configure them to run the service. While the *provisioning* is specific to the requirements of the deployment tool, the *Configuration* consists in installing and configuring any requirement specific to the service to deploy. For example the *Configuration* step could install a specific package needed by the software, or could configure a system configuration file for the specific use of the service.

h) Service install: Once the *Configuration* step is finished, the software services and their configuration files are set on the resources. Where services have to be deployed is known thanks to *Placement* and *Allocation*, and resources are prepared for installation thanks to the *Packaging*, *Provisioning* and *Configuration* steps. Finally the order of installation is known thanks to the *Dependencies* step.

i) Monitoring: Once the deployment is finished, the *Monitoring* step starts. Its aim is to monitor the physical and virtual resources to detect hardware or software failures.

j) Reconfiguration: Finally, the last step of the deployment process, which is also started after *Service install* is the *Reconfiguration*. As depicted in Figure 1, this step can rely on information from *Monitoring*. If needed, this step initiates a loop to one of the following steps: *Software packaging*, for example if a new version of a service is available; *Template*, each time a modification of the template is needed; *Placement*, for example if some resources fail; *Service install* if the deployment of a service as to be repeated without modification in other steps.

III. OpenStack deployment

In this section, five different existing solutions to deploy OpenStack in production are studied and compared: *Kolla*; *Enos*; *Juju*; *Kubernetes*; and *TripleO*. For each solution, a representation of the deployment process adapted from the model of Figure 1 is given. A new semantic is introduced in our new representations for each tool. This semantic is based on white and black circles that represent use and provide interfaces of components and their connection. The additional

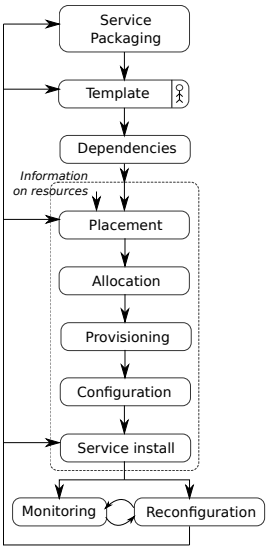

Fig. 1. Deployment model

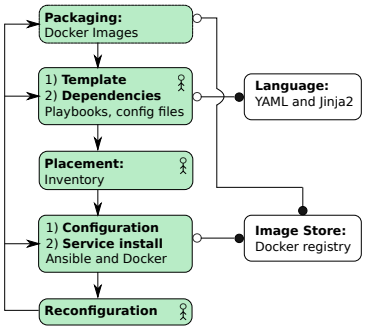

Fig. 2. Kolla model

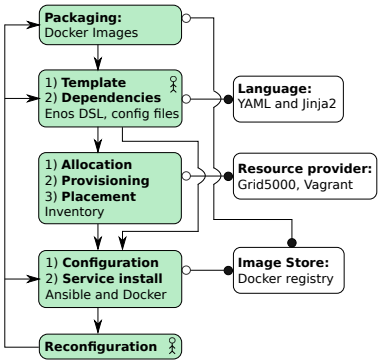

Fig. 3. Enos model

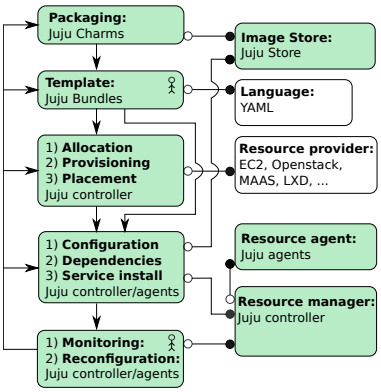

Fig. 4. Juju model

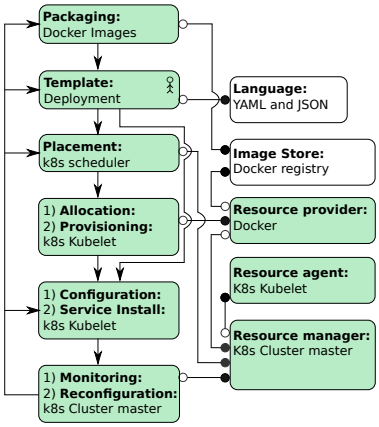

Fig. 5. Kubernetes model

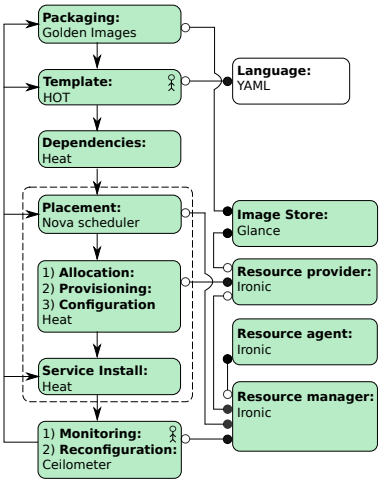

Fig. 6. TripleO model

components, compared to the generic model, give more details about each deployment phase regarding external tools. While internal tools are depicted by green boxes, external tools are represented by white boxes. Moreover, each deployment step can be handled *manually* (represented by a human icon on the figures) or *automatically* (default).

A. Kolla

Kolla is an OpenStack project that aims at deploying OpenStack by packaging each service as a Docker container [8]. It has to be noticed that Kolla is designed to deploy OpenStack and would not be natively used to deploy another distributed software. Kolla intensively relies on Ansible [9]. Ansible is a popular configuration management tool used to apply tasks on multiple hosts, described by a declarative language. The following details the Kolla deployment steps (according to the model defined previously). A representation of the deployment process of Kolla is depicted in Figure 2.

For the *Software packaging*, Kolla provides one *Dockerfile* per OpenStack service (more than 150). A *Dockerfile* describes the instructions to build container images. Kolla also provides a tool to easily build these images and to push them to a Docker registry as depicted in Figure 2.

The *Template* step consists in editing a configuration file which contains various deployment variables (*e.g.*, enabled services). These variables will be provided to Ansible playbooks to populate service configuration files written in Jinja2, a templating library that provides conditionals [10]. Neither relations or constraints are

specified by the operator. Service dependencies are manually defined in the Ansible playbooks, used to install services, which is the reason why they are represented in the same box as template in Figure 2. In Kolla, dependencies impact only the *Service install* phase.

Allocation and *Provisioning* are not supported by Kolla (thus, not represented on the figure). These phases must be performed by an external tool. Once the resources have been provisioned, the operator (the user) manually maps services to resource IP addresses through an inventory file, thus performing the *Placement* step manually. As a consequence, this is a particular situation where allocation and provisioning are realized prior the service placement. After the placement, Ansible is used first to generate the service configuration files, and to run the Docker containers according to Ansible playbooks (*Configuration* and *Service install* are in the same step). As depicted in the figure, container images are fetched from the registry. *Monitoring* is not supported in Kolla. However, software *Reconfiguration* can manually be triggered by the operator.

B. Enos

Enos is a framework to deploy and evaluate OpenStack on different infrastructures [11]. To that end, Enos extends Kolla with an intern provisioning engine that can request external resource providers. Moreover, to declare the desired topology, Enos leverages its own DSL which improves the template capacity of Kolla. Figure 3 represents the deployment model for Enos.

As depicted on this figure, the defined topology is used by the Enos engine to manage automatically the *Allocation*, then the *Provisioning* steps (which are thus merged in the same box in the figure). It has to be noticed that during the provisioning Enos handles the installation of Python and Docker on the resources. To that end, Enos defines internally a way to interface with an external resource provider, which is in charge of the resource *Allocation* and *Provisioning*. Currently, Enos is able to interface with Vagrant (deploying virtual machines), and the Grid'5000 testbed [12] (for bare metal provisioning). After the provisioning, Enos automatically generates the inventory files, thus performing an automatic *Placement* step. The rest of the deployment process is similar to the previous study.

C. Juju

Juju is the service orchestration tool developed by Canonical [13] and is meant to deploy any distributed software. Similarly to Enos, Juju relies on external providers, but its template is more expressive. Its deployment model is depicted in Figure 4.

In Juju, services are packaged as a set of files named *Charms*. A charm contains a collection of scripts which are triggered by Juju during the service lifecycle. These instructions (*e.g.*, install, start, stop) can be expressed by any scripting languages such as bash or Python. Juju charms can also define *provide* and *require* interfaces to describe relations or interactions between services.

Furthermore, bundles can express some constraints based on the resource capacity (*e.g.*, CPU cores, RAM capacity). Juju's community has written a large amount of charms and bundles that can be found on the Juju Store[2] (*e.g.*, OpenStack). Since service relations can be declared in the template, a *Dependency* graph is automatically generated by Juju. This dependency has an impact on the *Service install* step which interrupts the deployment of a service as long as its dependencies are not satisfied.

Similarly to Enos, the *Allocation*, *Provisioning* and *Placement* phases are automatic. The Juju client (CLI or GUI) is in charge of calling an external tool through the controller for *Allocating* and *Provisioning* machines. While classical cloud providers such as AWS or OpenStack can be used, more specific providers like LXD[3] or MAAS[4] can also be used to deploy charms on container or bare-metal machines. A Juju deployment is based on a Juju controller node which is in charge of the *Configuration* and the *Service install*. The Juju controller copies the charms to the appropriate resource and requests an agent deployed on it to trigger the required scripts (*e.g.*, install, start the services).

The particularity of Juju is that the controller keeps the states of agents in a database to monitor services and resources. As depicted in Figure 4, the operator (the user) can manually reconfigure the deployment when the template is modified (arrow to template), or modify the topology when scaling instructions are given (arrow to allocation), or even upgrade services (arrow to configuration).

D. Kubernetes

Spearheaded by Google, Kubernetes is a deployment tool for any containerized application [14]. Its main assets reside in the placement and monitoring phase. Indeed, Kubernetes contains a scheduler whose placement decisions are based on predicates (or constraints) and priorities policies (or objective functions). Furthermore, a master controller is in charge of the monitoring and the automated reconfiguration of the deployment. Its deployment process is represented in Figure 5.

Kubernetes' *Service packaging* is similar to Kolla, since each OpenStack service must be encapsulated with its dependencies in a dedicated Docker container.

[2]Juju Store: https://jujucharms.com/store

[3]The LXD container hypervisor https://www.ubuntu.com/cloud/lxd

[4]Metal as a Service https://maas.io/

However, Kubernetes does not provide a tool to build the related service containers.

Kubernetes templates are JSON or YAML-based files that describe specific Kubernetes objects (*e.g.*, pod, resources, replica sets). Compared to Juju, Kubernetes does not declare dependencies. Thus, this step is not depicted on Figure 5. However, hardware requirements and limitations can be specified. For example, a container can require at least 64MB of RAM, and be limited to 128MB.

In Kubernetes, a basic scheduling unit is called a *Pod*. A pod is a set of one or more containers that are scheduled on the same machine, and share the same resources (*e.g.*, IP address, storage volume). A pod is managed by a Kubelet agent. The *Placement* is automatically managed by the Kubernetes scheduler. It is in charge of placing unscheduled pods on nodes according to different predicates and priorities. First, predicates are used to filter out unsuitable nodes (*e.g.*, free resource requirements, hostnames, available host ports), then the scheduler applies priorities policies to rank the chosen ones (*e.g.*, favoring the least loaded nodes). To that end, the scheduler relies on the information stored at the resource manager, as depicted in Figure 5. The resource manager corresponds to the Kubernetes master. It requests information from the Kubelets about nodes, and stores this information in its database. Both predicates and priority functions are customizable, making the Kubernetes scheduler extensible. Once scheduled, the pod *Allocation* and *Provisioning* are automatically managed by Kubelets. The resource provisioner requires to load Docker images from a Docker registry (*e.g.*, DockerHub).

As depicted in Figure 5, the *Monitoring* phase relies on the cluster states stored in the master database mentioned earlier to trigger the appropriate reconfiguration if the template and the deployment state differ. For instance, failed containers are restarted or rescheduled for self-healing (respectively represented in the figure by the arrow to service install and placement) or auto-scaling, based on the scaling rules described in the template. Reconfiguration can also perform online rolling upgrades by swapping out containers which guarantees a continuous service delivery, and rollbacks to working containers if upgrading fails.

E. TripleO

TripleO is an OpenStack project that aims at deploying OpenStack instances using OpenStack itself (TripleO meaning OpenStack On OpenStack). To that end, an underlying OpenStack deployment (called *undercloud*) is necessary to provide the required services to deploy OpenStack instances on bare-metal machines. Since the undercloud is a pre-requisite, TripleO provides a tool to easily deploy undercloud services on a running host. Beyond classical OpenStack services, the undercloud requires the following OpenStack services: (i) Nova and Ironic for bare-metal *Provisioning*; (ii) Heat, the application orchestrator; (iii) Ceilometer for *Monitoring*; (iv) Glance, the image store. These projects are depicted in Figure 6 which represents the deployment model of TripleO. Heat has a major role in TripleO since it calls the undercloud's OpenStack API to deploy the overcloud services.

For the *Packaging*, TripleO deploys OpenStack services to bare-metal machines, encapsulated inside pre-built disk images. These so-called *golden images* contain an operating system, the desired OpenStack services and some tools for configuration. TripleO provides an image building tool, *diskimage-builder*, which aims at building the desired services on top of a native Linux distribution (*e.g.*, CentOS, Ubuntu). The built images can then be stored in Glance, the image store service.

HOT (Heat Orchestration Template) is a *Template* format used by Heat to orchestrate applications. It describes the required machines, their services, their configurations and their relations. Besides, it is possible to define hardware resource constraints, and scaling policies that will be used in the placement and monitoring phase respectively. The relations defined in the template are automatically used to compute a graph during the *Dependencies* step. This graph will be used at different levels: from the *Placement* step to the *Service install* phase, as depicted in Figure 6 by the dashed rectangle.

The *Placement* is automatically managed by the Nova scheduler which is in charge of mapping the services to the resources listed by Ironic (the resource manager in the figure). As depicted in Figure 6, Ironic both inspects the bare metal nodes (which is the role of the agent in the figure), and stores the collected data in its database (illustrated by the resource manager). The Nova scheduler works similarly to the Kubernetes scheduler described in the previous paragraph. It is based on filtering and weighting functions. Nova provides also a large set of predicates and priorities policies. The *Allocation* and *Provisioning* steps are requested by Heat, which relies on Ironic to provision bare metal machines with the golden images built during the *Service packaging* step, and stored in Glance.

As mentioned earlier, HOT can declare auto-scaling groups to manage service scalability based on resource monitoring (arrow to placement in Figure 6). Furthermore, rolling update policies can be defined in the template. The other reconfiguration aspects are currently manually managed by the operator.

	Kolla	Enos	Juju	Kubernetes	TripleO
Environment	containers	containers	any	containers	bare-metal
Packaging	containers	containers	scripts	containers	disk images
Template (all manual)	no relations no constraints no orchestration	no relations no constraints no orchestration	relations constraints no orchestration	no relations constraints orchestration	relations constraints orchestration
Dependencies	manual sequence install	manual sequence install	automatic graph install	manual sequence install	automatic graph from placement to install
Placement	manual extern/intern	automatic extern/intern	automatic extern/intern	automatic intern	automatic intern
Allocation	NS extern	automatic extern/intern	automatic extern/intern	automatic intern	automatic intern
Provisioning	NS extern	automatic extern/intern	automatic extern/intern	automatic intern	automatic intern
Monitoring	NS	NS	manual	manual/auto	manual/auto
Reconfiguration	manual template	manual template	manual resource software template scaling	manual/auto resource (auto) software (auto) template (manual) scaling (auto)	manual/auto resource (manual) software (manual) template (manual) scaling (auto)

TABLE I
COMPARATIVE TABLE OF THE STUDIED DEPLOYMENT TOOLS BASED ON THE STEPS DEFINED IN SECTION II

IV. DISCUSSION

Table I gives a comparative overview of the tools studied in the previous section, driven by the model steps described in Section II. In this table, a step that is *not supported* by a tool is tagged *NS*. In this section, we discuss various comparison criteria.

A. Environment heterogeneity

As depicted in Table I, by using the five deployment tools previously detailed, services can be deployed either on bare-metal or on virtualised environments (*i.e.*, containers or virtual machines). Each environment has its pros and cons. First, unlike VMs and bare-metal environments, Docker containers do not bundle a full operating system which makes them fast to deploy, scale and reschedule. This is an important consideration for dynamic infrastructures used for instance for IoT and mobile edge applications, where nodes can join, leave or move. On the other hand, Docker containers are not suited for any kind of service. Contrary to bare-metal and VM environments, Docker containers are designed for stateless services, and are not mature enough to run databases that are in charge of persistent data [15]. Furthermore, while VMs share the underlying hardware between multiple operating systems, a bare-metal environment monopolizes all of the physical resource of the machine, which provides more hardware resources and reliable capacity for services that are sensitive to performance. Thus, by managing heterogeneous environments, a deployment tool is able to place services to the appropriate environment, according to their nature and requirements. For instance, stateless services can be deployed on Docker containers for scalability and reconfiguration, while stateful and I/O intensive databases could be deployed on bare-metal servers. Moreover, the deployment tool can package a service for the appropriate resource. For instance in IoT infrastructures, virtual machines might not be deployable on small Things (devices) with very limited capacity, while Docker containers might fit. In our comparison, Juju is the only one able to manage the three types of environment by using scripts and external providers.

B. Size of the service packages

Three different kinds of service packages are represented in Table I: *Scripts*; *Container images*; and *Disk images*. Packages are characterised by the time required to be built, and by their associated size (in bytes). They have an impact on the time spent for *Packaging*, *Service install* and for *Reconfiguration*. The smaller packages are, the faster they are transferred to the *Image Store*, and to the resources during the *Service install* step. Since they do not embed a full operating system (but libraries and services) containers are faster to build and transfer than disk images. TripleO is the only one which is limited to disk images.

C. Expressiveness of the template

The different *Template* languages offered by deployment tools differ in the way they can express relations, constraints and orchestration rules like scaling or reconfiguration policies. An expressive language is required to manage complex systems and resources since it enables better *Dependencies*, *Placement* and *Reconfiguration* processes. However, it increases the complexity of writing templates since fine-grained descriptions must be

provided. For example, the operator must indeed clearly understand the services and the application workflow to define dependencies. For instance, Kolla, Enos and TripleO are straightforward since they are dedicated to OpenStack. For instance, in Kolla and Enos, the workflow is provided in the Ansible playbooks. Thus, the operator only needs to select which OpenStack services to deploy. However, they have limitations related to the lack of expressiveness to declare orchestration rules. However, Kubernetes and HOT, the template language used in TripleO, can express such rules.

Dependencies can be limited to a *sequential* order or can be represented as a complete *dependency graph*. Moreover, dependencies can be used to optimize one or a set of the following steps: *Placement*; *Allocation*; *Provisioning*; *Configuration*; *Service install*. HOT is the only template language declaring service dependencies to build a dependency graph that improves all these steps.

D. Automatic placement/allocation/provisioning

As depicted in Table I, *placement*, *allocation* and *provisioning* steps can be manual or automatic, and managed internally or externally. While automatic management is required to ease the deployment, it requires to develop a placement manager that can be more or less complex. For instance, the placement manager of Enos is straightforward, while Kubernetes and TripleO rely on extensible schedulers. On the one hand, complex placement constraints and objective functions like locality, energy considerations and capacity awareness might be required for some deployments. In such case, an internal management of these steps brings control and fine-grained knowledge on resources. Kubernetes and TripleO are the only tools that can express such constraints, and that can retrieve fine-grained description of resources due to their agents. Note that in such case, the placement step is done after the allocation and the provisioning. On the other hand, when these steps are managed externally, like in Enos and Juju, the tool is able to deploy on multiple clouds (*e.g.*, AWS, GCE), and the placement is done after the allocation and the provisioning.

E. Automatic reconfiguration and prerequisites

Monitoring and *Reconfiguration* steps are directly linked and can handle multiple cases:

- *resource* monitoring corresponds to reconfigurations due to failures or joining/leaving resources;
- *software* monitoring is associated to reconfigurations when services fail;
- *template* monitoring, refers to the reconfiguration when the template and the current deployment differ (*e.g.*, updates, template modifications);
- *scaling* monitoring triggers reconfiguration when specific events happen (*e.g.*, overloaded resources).

An architecture based on controller and agents is required for monitoring and automatic reconfiguration. Agents must be installed on resources to send metrics to the controller. Juju, Kubernetes and TripleO are such deployment tools which require that the controller and agents are deployed, raising a chicken and egg issue. In our study, we have indeed observed some prerequisites. Juju needs a controller on the desired cloud provider, Kubernetes requires a controller and the Kubelet agents, and TripleO requires a running undercloud to deploy its overclouds.

F. IaaS as a deployment tool

The deployment process depicted in Figure 1 is a subpart of an IaaS managing system. It has been illustrated by TripleO in Figure 6 since TripleO relies on the OpenStack services of the undercloud to deploy overclouds. Thus, a deployment tool like TripleO would directly benefit from the improvements of OpenStack services, factorizing the effort for both applications (OpenStack as an IaaS and as a deployment tool). For instance, an improvement in the Nova scheduler would improve the placement step of the deployment model.

G. Discussion conclusion

The conclusion of this comparison is that each tool has its pros and cons. Nevertheless, TripleO is a particularly interesting candidate. While it manages only bare-metal machines, TripleO can be improved to manage OpenStack services that handle containers and virtual machines. Its template has a good expressiveness and is able to define constraints, dependencies and orchestration rules. Its placement management is based on the nova scheduler which is extensible, and Ironic agents to collect resource metrics. Finally it can benefit from the ongoing research effort for a fully decentralized OpenStack system to provide a deployment tool adapted for the future dynamic and global infrastructures.

V. Related work

This section presents existing surveys that are related to the deployment of distributed modular softwares. Each related survey presented in this section can be attached to one or multiple steps of the deployment process presented in Figure 1.

First the survey presented in [16] focuses on cloud modeling languages. Studied languages have not been used to deploy OpenStack and cannot be compared to the one presented in this paper. One can note, though, that the TOSCA [17] modeling specification is close to HOT [5] and is described in the survey [16].

[5]http://fr.slideshare.net/openstackil/heat-tosca

Secondly, while the title of [18]: "Resource Provisioning Techniques in Cloud Computing Environment - A Survey" refers to resource provisioning, it can be classified into the *Placement* deployment step presented in Figure 1. The survey [19] also refers to the placement step of the deployment process, however it studies the specific case of placement algorithms in federated clouds, *i.e.*, a federation of IaaSes. Furthermore, the work presented in [20] is a survey on the *Placement* and *Allocation* steps of our deployment process. These three surveys compare scheduling algorithms.

Thirdly, the work presented in [21] focuses on the *Configuration* step of the deployment process. One can note that the configuration step of the deployment process can itself be considered as a deployment process at a lower level (closer to resources). For this reason, some steps or criteria are common to our contribution.

Finally, the last category of related survey deals with the *Monitoring* step. The survey presented in [22] introduces a taxonomy of seven different levels where cloud monitoring could appear. The paper [23] also proposes a survey on cloud monitoring which relies on a heavy set of categories including for example scalability, portability, flexibility or resource usage metering.

To conclude, many existing surveys related to deployment have studied a specific subpart of a deployment process. Our survey, on the other hand, introduces an overall deployment process on which our comparison and analysis is based. Moreover, as far as we know, no existing survey has clearly compared production deployment tools, especially concerning the specific case of OpenStack. To be able to handle large scale and highly dynamic future infrastructures such as Fog and Edge, though, OpenStack appears as one good candidate [24].

VI. Conclusion

In this paper, we have first analyzed the deployment phases through a model in ten steps to fully understand the deployment process. This model has then served to analyze and compare five existing deployment tools used to deploy OpenStack in production (*i.e.*, Kolla, Enos, Juju, Kubernetes and TripleO). The conclusions of this survey paper are that each of the five studied tools have their own pros and cons. While Kolla, Enos and Juju are designed to be easy to operate, Kubernetes and TripleO are the most sophisticated tools in terms of deployment features. Finally, we considered OpenStack since it is a complex modular application, but our model can be applied on any distributed software.

Acknowledgments

Supported by the Inria Project Lab program Discovery: an Open-Science Initiative aiming at implementing a fully decentralized IaaS manager: http://beyondtheclouds.github.io.

References

[1] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002, ISBN: 0201745720.

[2] M. van Steen, G. Pierre, and S. Voulgaris, "Challenges in very large distributed systems," *Journal of Internet Services and Applications*, vol. 3, no. 1, pp. 59–66, 2012, ISSN: 1869-0238.

[3] *Deployment and Configuration of Component-based Distributed Applications Specification*, http://www.omg.org/spec/DEPL/4.0/PDF, Object Management Group, Inc., Apr. 2006.

[4] *OpenStack*, https://www.openstack.org/.

[5] T. Bell, B. Bompastor, S. Bukowiec, *et al.*, "Scaling the CERN OpenStack Cloud," *Journal of Physics: Conference Series*, vol. 664, no. 2, 2015.

[6] F. Hermenier and S. Demassey, "BtrPlace: Flexible VM Management in Data Centers," in *Conference on Optimization & Practices in Industry, PGMO-COPI'14*, France, Oct. 2014.

[7] P. Silva, C. Pérez, and F. Desprez, "Efficient Heuristics for Placing Large-Scale Distributed Applications on Multiple Clouds," in *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'16)*, Colombia, May 2016.

[8] *Kolla*, https://github.com/openstack/kolla.

[9] *Ansible*, https://www.ansible.com/.

[10] *Jinja2*, http://jinja.pocoo.org/.

[11] R.-A. Cherrueau, A. Lebre, D. Pertin, *et al.*, "ENOS: a Holistic Framework for Conducting Scientific Evaluations of OpenStack," Inria, Nantes, Technical Report RT-485, Nov. 2016.

[12] D. Balouek, A. Carpen Amarie, G. Charrier, *et al.*, "Adding Virtualization Capabilities to the Grid'5000 Testbed," in *Cloud Computing and Services Science*, vol. 367, Springer International Publishing, 2013, pp. 3–20, ISBN: 978-3-319-04518-4.

[13] *Juju*, https://jujucharms.com/.

[14] *Kubernetes*, http://kubernetes.io/.

[15] M. Chinkov, *Why databases are not for containers?* https://myopsblog.wordpress.com/2017/02/06/why-databases-is-not-for-containers/, Jan. 2017.

[16] A. Bergmayr, M. Wimmer, G. Kappel, *et al.*, "Cloud modeling languages by example," in *7th IEEE International Conference on Service-Oriented Computing and Applications, SOCA 2014, Matsue, Japan, November 17-19, 2014*, 2014, pp. 137–146.

[17] *Topology and Orchestration Specification for Cloud Applications Version 1.0*, http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html, 2013.

[18] N. Bhavani and S. Guruprasad, "Resource provisioning techniques in Cloud computing environment - a survey," *IJRCCT*, vol. 3, 2014, ISSN: 2278-5841.

[19] M. Gahlawat and P. Sharma, "Survey of virtual machine placement in federated Clouds," in *2014 IEEE International Advance Computing Conference (IACC)*, Feb. 2014, pp. 735–738.

[20] V. P. Anuradha and D. Sumathi, "A survey on resource allocation strategies in Cloud computing," in *International Conference on Information Communication and Embedded Systems (ICICES2014)*, Feb. 2014, pp. 1–7.

[21] T. Delaet, W. Joosen, and B. Vanbrabant, "A survey of system configuration tools," in *Proceedings of the 24th International Conference on Large Installation System Administration*, ser. LISA'10, San Jose, CA: USENIX Association, 2010, pp. 1–8.

[22] G. Aceto, A. Botta, W. de Donato, *et al.*, "Cloud monitoring: A survey," *Computer Networks*, vol. 57, no. 9, pp. 2093–2115, 2013, ISSN: 1389-1286.

[23] K. Fatema, V. C. Emeakaroha, P. D. Healy, *et al.*, "A survey of Cloud monitoring tools: Taxonomy, capabilities and objectives," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2918–2933, 2014, ISSN: 0743-7315.

[24] A. Lebre, J. Pastor, A. Simonet, *et al.*, "Revising OpenStack to Operate Fog/Edge Computing infrastructures," in *IEEE International Conference on Cloud Engineering*, Vancouver, France, Apr. 2017.