COMPUTING SCIENCE

Dynamic Deployment of Scientific Workflows in the Cloud using Container Virtualization

Rawaa Qasha, Jacek Cala, Paul Watson

TECHNICAL REPORT SERIES

No. CS-TR-1501 October 2016

TECHNICAL REPORT SERIES

No. CS-TR-1501 October, 2016

Dynamic Deployment of Scientific Workflows in the Cloud using Container Virtualization

Authors: Rawaa Qasha, Jacek Cala, Paul Watson

Abstract:

Scientific workflows are increasingly being migrated to the Cloud. However, workflow developers face the problem of which Cloud to choose and, more importantly, how to avoid vendor lock-in. This is because there are a range of Cloud platforms, each with different functionality and interfaces. In this paper we propose a solution system that allows workflows to be portable across a range of Clouds. This portability is achieved through a new framework for building, dynamically deploying and enacting workflows. It combines the TOSCA specification language and container-based virtualization. TOSCA is used to build a reusable and portable description of a workflow which can be automatically deployed and enacted using Docker containers.
We describe a working implementation of our framework and evaluate it using a set of existing scientific workflows that illustrate the flexibility of the proposed approach.

Printed and published by Newcastle University,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

Dynamic Deployment of Scientific Workflows in the Cloud using Container Virtualization

NEWCASTLE UNIVERSITY
Computing Science. Technical Report Series. CS-TR-1501

Abstract:

Scientific workflows are increasingly being migrated to the Cloud. However, workflow developers face the problem of which Cloud to choose and, more importantly, how to avoid vendor lock-in. This is because there are a range of Cloud platforms, each with different functionality and interfaces. In this paper we propose a solution â€“ a system that allows workflows to be portable across a range of Clouds.
This portability is achieved through a new framework for building, dynamically deploying and enacting workflows. It combines the TOSCA specification language and container-based virtualization. TOSCA is used to build a reusable and portable description of a workflow which can be automatically deployed and enacted using Docker containers.
We describe a working implementation of our framework and evaluate it using a set of existing scientific workflows that illustrate the flexibility of the proposed approach.

About the authors

Professor Paul Watson is currently a Professor in School of Computing Science, Newcastle University. Paul is the Director of the Digital Institute and directs the £12M RCUK-funded Digital Economy Hub on Social Inclusion through the Digital Economy and is PI of the EPSRC Centre for Doctoral Training in Cloud Computing for Big Data. His research interest is in scalable information management with a current focus on Cloud Computing. Paul is a Chartered Engineer, a Fellow of the British Computer Society, and a member of the UK Computing Research Committee. Paul graduated in 1983 with a BSc in Computer Engineering from Manchester University, followed by a PhD on parallel graph reduction in 1986. In the 80s, as a Lecturer at Manchester University, he was a designer of the Alvey Flagship and Esprit EDS systems. From 1990-5 he worked forICL as a system designer of the Goldrush MegaServer parallel database server, which was released as a product in 1994.

Dr Jacek Cala is currently a Research Associate in the Scalable Computing group, School of Computing Science at Newcastle University. Jacek joined in 2009 and is involved in e-Science Centre project under Professor Paul Watson. His main role is to investigate possible applications of Microsoft cloud computing technologies in the ESC. Jacek received his MSc in Computing Science from the AGH-University of Science and Technology (Cracow, Poland) in 2001. Between 2001-2009 he worked at the AGH-UST as a Teaching and Research Assistant in Distributed System Research Group. He was involved in many interesting research projects related to telemedicine, adaptive distributed systems, distributed management, virtual laboratories.

Jacek's main present and past research interests are related to the areas of distributed systems, component-based systems, compositional adaptation, mobile systems and more recently cloud computing.

Rawaa Qasha is a 4rd year PhD student at the school of Computing Science, Newcastle University, UK. Her PhD research is on automatic deployment and reproducibility of workflow in the Cloud. She received the master degree from Computer Sciences department, University of Mosul, in 2000. She is also a lecturer (assistant professor) in computer science at University of Mosul, Iraq. Her research interests concentrate on Cloud computing, distributed system, workflow deployment and reproducibility, container virtualization, E-Science system.

Suggested keywords

Dynamic Deployment, Scientific Workflows, Cloud Computing, Container Virtualization, TOSCA

Dynamic Deployment of Scientific Workflows in the Cloud using Container Virtualization

Rawaa Qasha*†, Jacek Cała*, Paul Watson*,
*School of Computing Science, Newcastle University, UK
†College of Computer Sciences and Mathematics, Mosul University, Iraq
Emails: {r.qasha, jacek.cala, paul.watson}@newcastle.ac.uk

***Abstract*—Scientific workflows are increasingly being migrated to the Cloud. However, workflow developers face the problem of which Cloud to choose and, more importantly, how to avoid vendor lock-in. This is because there are a range of Cloud platforms, each with different functionality and interfaces. In this paper we propose a solution – a system that allows workflows to be portable across a range of Clouds.**

This portability is achieved through a new framework for building, dynamically deploying and enacting workflows. It combines the TOSCA specification language and container-based virtualization. TOSCA is used to build a reusable and portable description of a workflow which can be automatically deployed and enacted using Docker containers.

We describe a working implementation of our framework and evaluate it using a set of existing scientific workflows that illustrate the flexibility of the proposed approach.

I. Introduction

In recent years, Cloud Computing has gained remarkable momentum in both academia and industry. An increasing range of applications are now being developed in the cloud, and this includes scientific workflow systems [1].

Workflows are frequently used in scientific research to orchestrate the execution of complex experiments on distributed resources. A workflow can be considered as a model defining the structure of the computational and/or data processing tasks necessary for the management of a scientific process [2]. One key reason for their adoption is that they offer the opportunity to share, exchange and reuse services and experimental methods [3].

The scalability and ability to acquire resources on-demand offered by Cloud Computing makes it attractive for workflow management [4]. However, efficiently meeting workflow requirements in the cloud requires addressing key issues in the provisioning of the execution environments, and subsequent workflow execution [1]. Due to the rapid evolution of existing cloud platforms, and the emergence of new providers, one very important challenge is in making workflows portable and reusable across different cloud platforms.

This is important for several reasons: it avoids Cloud vendor lock-in, mitigates the risk of a cloud vendor failing and enables users to switch to a cheaper cloud. Also, for a scientific method to be effectively reused over time, and for experiments to be reproduced, the repeatability of workflow deployment and configuration steps is crucial. Experience has shown that if workflow deployment and configuration steps cannot be easily repeated, then the value of the workflow as a way to share and reproduce scientific results is quickly lost [5].

This paper describes a new solution to this problem: a method for automatic deployment of workflows on the Cloud. We extended our modeling approach proposed earlier in [6] and implemented the provision and deployment of workflows in a way that significantly increases their portability.

A major advantage of scientific workflows is the abstract way in which they can combine together a set of different tasks to encode a single analysis. Often, however, these tasks are heterogeneous components each with their own set of dependencies. For example, different workflow's tasks may need the same library with different versions or each task to be executed on a specific version of the operating system. This poses a serious challenge in the description and deployment of workflows. Thus, the workflow descriptor needs to include not only the abstract graph of interconnected tasks but also, so often ignored, details of component implementation and deployment. Moreover, a robust deployment facility should support the isolation of component execution to ensure minimal interference between them.

To address these challenges we present a new framework to describe, build, dynamically deploy and enact workflows on the Cloud. The framework integrates the OASIS standard: Topology and Orchestration Specification for Cloud Applications (TOSCA) [9] and container-based virtualization. TOSCA supports the description of Cloud applications in a portable way [10], which we exploit to allow heterogeneous workflows to be deployed in the Cloud. Container-based virtualisation offers the opportunity for rapid and efficient building and deployment of lightweight workflow components [8]. In this work, we use Docker[1] containers to dynamically provision the execution environment and construct the full software stack required by a workflow component or group of components. Both, TOSCA and Docker allow us to improve the reusability and reproducibility of workflow-based applications.

To demonstrate our approach in practice we model a set of scientific workflows using TOSCA, automate their deployment and dynamically provision their execution environment using containers implemented in Docker. Our examples involve typical scientific workflows with data dependencies between tasks creating a directed acyclic graph. We adopted TOSCA

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

to represent not just the workflow itself but also its components, library dependencies and the configuration of the whole workflow-based application including its hosting environment. The framework is generic enough to cover a variety of scenarios which we used for evaluation in the following sections.

In this paper we present a significant development of our previous work described in [6]. We introduce the following new contributions:

- we show how the modeling approach proposed in [6] can be used to describe and implement the provisioning and deployment of workflows across different Cloud infrastructures, thereby ensuring application portability.
- we show that by using our framework we can construct and dynamically deploy the full software stack required by a workflow component or group of components.
- we exploit container-based virtualization to improve deployment portability and isolate the execution of heterogeneous workflow components.
- we show how our framework supports a range of deployment options for efficiency and security isolation.

II. Related Work

For over a decade, scientific workflows have been a successful method to encode and repeat *in-silico* scientific experiments and a vast number of platforms and languages exist to model workflows [11]. However, most scientific Workflow Management Systems (WfMS) focus on workflow expressiveness and ease of modeling. Only a few solutions such as e-Science Central [12], Pegasus [13], Galaxy [14] and, more recently, HyperFlow [15] tackle the problem of scientific workflow deployment in a distributed environment.

e-Science Central (e-SC)[2] is a cloud-based WfMS that provides capabilities to store, analyse and share data among scientists. It includes a workflow enactment engine to which users can submit their workflows via a web browser or an external application. The system implements a simple dataflow model in which a workflow comprises a set of interconnected blocks. Blocks can be of different types (Java, R, Octave, etc.) and the definition of a block also contains software dependencies that must be met to start it. Before running a block, any unavailable libraries are downloaded from the server on demand, and then the engine start executing the block. That makes the e-SC workflow engine generic and independent of the workflows it is to enact.

Pegasus is a well-established WfMS. It allows workflows to be defined as abstract and resource-independent, and later, before execution, they are mapped by the system into concrete, platform-specific execution plans. The plans are enacted by HTCondor DAGMan that tracks dependencies and releases tasks as they become ready, whilst HTCondor Schedd runs them on available resources. To look up user executable files that implement workflow tasks Pegasus uses the *Transformation Catalog*. The catalog maps tasks into executables specific to the underlying execution environment whether it is HTCondor pool, HPC or Cloud. However, automatic installation or deployment of executable files is limited to Pegasus auxiliary executables, whilst the Transformation Catalog supports only the discovery of user executables.

Recently, the Galaxy WfMS has attracted attention, especially in the bioinformatics domain. Much like most other WfMS, it relies on external tools and datasets, and to facilitate their installation it uses Galaxy ToolShed and Data Managers. But the execution of workflows in Galaxy is considered separate from the installation of dependencies, making workflows usable only if all the dependencies are available prior to execution. To alleviate this issue the Galaxy team have offered dedicated data and VM cloud images with a suite of the most common bioinformatics tools and data [16].

Unfortunately, solutions like Galaxy, which are based on VM images, require significant maintenance effort when users need to add new, or update existing application tools over time. This requires rebuilding the images, a task rarely supported by the WfMS itself. Conversely, using our framework, updates of any task or dependency library can be achieved easily through updates in the TOSCA description, which then allows our system to provide automated, on-demand provisioning of the updated artifacts. Also, Docker enables images with the updated contents to be captured during the deployment process which gives the ability to create the images automatically.

Most of the existing WfMSes use a very specific workflow definition language which limits their portability. An effort to design a common language for scientific workflows (CWL) has recently been started,[3] yet it is at the early stage and not mature enough for practical applications. Also, despite its intention to provide a generic description for workflow processes and their dependencies, CWL does not include the description of the execution environment, nor the dependencies required to execute tasks. Instead, it relies on Docker as a mechanism to capture the installation and execution of tasks and dependencies.

Instead, what we demonstrate, is a method to define scientific workflows in a comprehensive and portable way. Firstly, adopting the TOSCA standard ensured that a workflow definition is portable and can be deployed and executed in a TOSCA-compliant runtime environment such as Cloudify[4], and potentially OpenTOSCA [17]. Secondly, by using TOSCA we can describe a workflow together with the complete software stack necessary to run its components – including virtual machines, containers and any dependency libraries. Thirdly, we adopt Docker as an optimisation mechanism that can improve deployment – our workflows and tasks can be deployed and enacted using only the TOSCA description, whilst Docker images, if exist, speed up the installation phase.

Currently, most of the efforts to use TOSCA, such as [18], [19], [20], are focused on the exploration of possible applications of the standard to manage various types of distributed

[2]http://esciencecentral.co.uk

[3]https://dx.doi.org/10.6084/m9.figshare.3115156.v1

[4]http://getcloudify.org

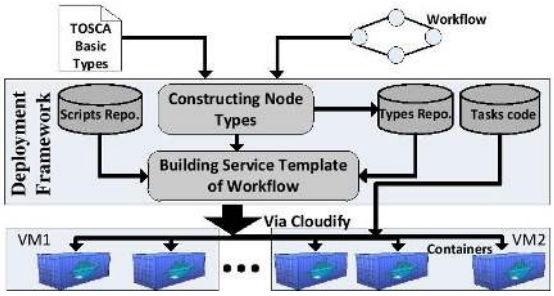

Fig. 1. Steps from the definition to enactment of a workflow.

systems on the cloud. None of these, however, has tried to use TOSCA in scientific workflow enactment.

Virtualization techniques based on containers have emerged in the last years as an alternative to hypervisor-based virtualization [7]. The key reasons for their adoption in application deployment [8] and also in the Cloud [21] are: the reduction in resource usage, the rapid provisioning features and good execution isolation features that prevent a single container from consuming all available resources. Also a number of authors have used Docker to package and provision applications and middleware recently (cf. [21], [22]). As for deploying scientific workflows using Docker, apart from the CWL mentioned above another recent solution is Skyport [23]. In Skyport, however, all tasks images are created manually and then used in the deployment process depending on the system specific meta-data. Instead, we propose a framework that combines TOSCA and Docker and can dynamically inject into a container the full software stack required to execute the complete workflow or each of the workflow tasks separately.

III. Scientific Workflow Deployment Framework

Our framework for the deployment and enactment of workflows is depicted in Fig. 1. It has been implemented as a set of the reusable components and packages that reside in our software repositories, so they can be used by the workflow execution node and also shared between users.

First, to build a workflow, we follow the TOSCA-based approach proposed in our previous work [6], and prepare basic workflow components: Node and Relationship Types, and then a Service Template which includes Node and Relationship Templates. Types are used to describe workflow components (tasks and their dependencies), whereas the Service Template describes the overall structure of the workflow. It contains Node and Relationship Templates to denote all the instances of software components, library dependencies and the execution environment together with the container and VM.

Next, in the template we also include the lifecycle management scripts and references to software artifacts. The scripts implement deployment actions of workflow tasks and are available in our lifecycle Scripts Repository. The software artifacts include the actual code that implements workflow tasks; these can be task-specific files and executables or Docker images that encapsulate one or more tasks with their dependencies. The artifacts are stored in our Task Code Repository to be reused across different tasks and workflows.

Finally, to deploy and enact a workflow we submit the Service Template together with scripts and artifacts to a TOSCA runtime environment. Although in this paper we assume that before the submission users have Cloudify and Docker installed, we have also developed a *one-click deployment script* so that they can easily enact a workflow on a clean, pure-OS VM in the Cloud. The script starts a multi-steps process that installs and configures basic prerequisites, such as Docker and Cloudify, and then initiates the execution of the workflow.

The following sections present details of the three steps described and discuss the data exchange mechanism implemented by the framework.

A. Building the Workflow Topology

Usually, modelling of scientific workflows focuses merely on the horizontal dimension – the data dependencies between tasks – whereas aspects related to the vertical dimension, such as creating tasks' runtime environment, remain ignored. They are crucial, however, to improve workflow portability and reproducibility. In our framework, we use TOSCA to model the structure of a workflow in both dimensions that can span both the horizontal space and vertical stack of software components. In the horizontal dimension, components rely on each other when they need to communicate and exchange data. In the vertical dimension, they are dependent as in the host-hosted relationship, where the host component provides an execution environment for the hosted component.

Building a workflow using TOSCA starts by defining Node and Relationship Types. A Node Type declares properties and lifecycle interfaces of a workflow component (task, library and container). These include the task name, version and a URL to task artifacts, as well as task configuration parameters. A Relationship Type can define a horizontal dependency between tasks and vertical host-hosted relationship between workflow components and their containers.

Given the types, the Service Template of the workflow is constructed as a graph of Node and Relationship Templates which represent specific instances of the types. If types declare properties and interfaces, templates provide values for the properties and implement lifecycle interface operations using scripts. The structure of the workflow is included in the *topology* part of the Service Template which the TOSCA runtime can analyse to build a step-by-step sequence of deployment operations. And although TOSCA was intended to describe service-based systems, we impose data dependencies and sequential enactment of tasks by using the dependency relationship between components.

Importantly, the Service Template includes not only the high-level structure of the workflow (i.e. task dependencies) but also all library dependencies and the definitions of container and virtual machine that are supposed to host the workflow components. Thus, we can capture the complete software stack required to deploy and enact the workflow.

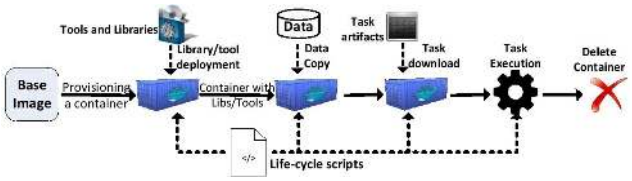

Fig. 2. Isolated deployment of a workflow task.

More details about how we use TOSCA to define scientific workflows can be found it our previous work [6].

B. Managing the Workflow Deployment Lifecycle

Types and templates consist of essential sub-elements that cover their lifecycle – operations attached to nodes and relationships. These operations are used to create, configure and start services and also to pre- and postconfigure relationships. We implemented them as a set of generic scripts that can:

- initialize a shared space to exchange data between tasks,
- fetch the input data files required to run a task,
- provision the host environment (a container) using an image specified in the workflow Service Template,
- configure the required library dependencies,
- download, configure and start a workflow task, and
- transfer data between tasks running either in a single or multiple VMs.

As these scripts are reusable across a range of workflows and tasks, we store them in our Lifecycle Script Repository,[5] so they can be easily included in any newly designed workflows. We refer to this repository in all our example workflows that we used to evaluate the framework.

C. Task Deployment

Once the workflow Service Template, lifecycle management scripts and all task artifacts are prepared, we can submit our workflow to a TOSCA runtime environment for deployment and enactment. Given the Service Template, a TOSCA runtime environment can deploy and execute tasks one by one in the order implied by the relationships between nodes. Each workflow task follows the deployment process shown in Fig. 2.

First, a Docker container is created using a task image indicated in the Service Template. The image may be generic, available from the Docker Hub[6] or it may be generated by the user and may include libraries and software required by the task(s). If more than one task is designated to run in a specific container, the container is created once and then reused by all of the tasks; this is possible regardless of the task order. For example, a sequence of workflow tasks: $T_1 \rightarrow T_2 \rightarrow T_3$ can be deployed such that T_1 and T_3 are hosted in one container, whereas T_2 is hosted in another one according to tasks' dependencies and isolation requirements. Then, the framework will maintain the appropriate order of execution while reusing the first container to run task T_3. That gives the workflow designer freedom in planning how tasks are distributed across containers without affecting runtime effectivness.

Once the task container is running, the installation of dependency libraries takes place according to their order in the Service Template. Depending on the initial contents of the selected Docker image, this may involve the installation of some software required to run the task. In the next step task artifacts are installed in the container and that is followed by copying input data required to run the task.

Note that if the image already includes all required dependencies and artifacts, no installation or copy operation is needed. Usually, however, the task artifacts will need to be downloaded from our Task Code Repository. This allows us to freely update tasks and minimize the number of specific Docker images held in the repository.

Finally, with all prerequisites in place the task execution is initiated, and upon its completion the output data are transferred out of the container. If the completed task is the last task to be executed in that container, the container is also terminated and deleted.

D. Data Transfer

Before the submission of the Service Template, the user needs to identify the input data that the workflow is going to process. Input data could be fetched from external repositories if needed but, in this paper, we use the host VM disk as a space to store input/output data. We also use the host disk as a shared space to exchange data files between tasks. In that way we can minimise overheads related to transferring inputoutput files and data between tasks deployed on the same machine.

This is only one possible method to exchange data – another is via a direct network connection between tasks, which we use in the case of tasks deployed over multiple VMs. Finally, the system can use Cloud-based data repositories such as Amazon S3 and Azure Blob Store, for the case of multi-VM deployments across different Clouds.

Importantly, the process of transferring data between tasks is performed automatically by the lifecycle script that implements inter-task dependency relationship and it remains hidden from the workflow designer.

IV. Experiments and Evaluation

To validate our approach, we conducted a set of experiments in which we deployed selected workflows, originally created in e-Science Central. The aim was to investigate and analyze several aspects concerned with the performance of the proposed design and deployment method. First, we wanted to measure time required to deploy workflows in local and public Cloud environment. We also wanted to see the overheads related to the deployment of workflows using single- and multi-container strategies. Additionally, we compared the impact of the use of generic and prepackaged Docker images on the overall workflow execution time.

All the experiments presented here are based on workflows and tasks that are publicly available in our GitHub reposito-

[5]https://github.com/WorkflowCenter-Repositories/Core-LifecycleScripts
[6]https://hub.docker.com

ries.[7] Although in this paper we focus our discussion on a single VM host with multiple containers, our framework can be also used to deploy workflow tasks on different VMs.

A. Experimental Setup

To illustrate that the approach is generic, the workflows we used for the performance evaluation vary in terms of structure, the number of tasks and their dependency libraries. Table I summarizes the basic properties of the workflows.

TABLE I
WORKFLOWS SELECTED TO TEST OUR DEPLOYMENT APPROACH.

Workflow Name	No. of tasks	Dependency libraries
Neighbor Joining (NJ)	11	ClustalW, MegaCC, Wine Java, Core-lib
Sequence Cleaning (SC)	8	SAMTools, Java, Core-lib
Random A (RA)	7	Java, Core-lib
Random B (RB)	3	Java, Core-lib

The *Neighbor Joining* workflow (NJ) is a pipeline used in the EUBrazil Cloud Connect project[8] to perform species identification of Leishmania parasite and sandflies using the neighbor-joining method. It consists of 11 tasks of which nine are Java-based and two other (Clustal and MEGA-NJ) wrap existing executable tools. The MEGA-NJ task is a Windows executable and to be executed in Linux it requires the Wine library. The *Sequence Cleaning* workflow (SC) is one of the steps in the Next Generation Sequencing pipeline implemented in the Cloud-e-Genome project [24]. It consists of eight tasks of which seven are Java-based and one is a wrapper around the SAMTools executable. Finally, *Random A* and *B* are simple workflows that consist of only Java tasks to inverse matrix and compress/decompress files, respectively.

To provision the execution environment for workflow tasks we used different Docker images to run the containers (Table II). The *Ubuntu:14.04* and *CentOS* images are pure OS images pulled from the Docker Hub and do not contain any tools used by the workflows. The *Basic* image contains a set of common tools used by our solution, such as Java and wget. For two selected workflows: NJ and SC we also prepared two specialized images: *CompleteNJ* and *CompleteSC*, respectively. These images extended the *Basic* image with all additional tools, libraries and task artifacts required to run each task in the workflow.

Although our framework allows us to provision containers on different VMs, all containers used throughout the experiments were deployed in a single virtual machine. We used Cloudify version 3.1 and its CLI to run the workflow blueprint file (the Service Template). To manage containers we used Docker version 1.5.0.

B. Experiment 1: Deployment and Enactment Time

In the first experiment we compared the deployment and enactment time of the test workflows on different execution environments. Each workflow task was running in a separate container with the ability to use a different image. We used the Basic and CentOS images for *Neighbor Joining* and the Basic image for the other three workflows. We also used a local VM and two public Cloud providers to host the Cloudify runtime, as presented in Table III.

TABLE II
DOCKER IMAGES USED IN THE EXPERIMENTS.

Image name	Contents	Image size [MB]
Ubuntu:14.04	as in the Docker Hub	188
CentOS	as in the Docker Hub	178
Basic	Ubuntu:14.04 + Java + wget	561
CompleteNJ	Basic + all NJ deps. + blocks	1536
CompleteSC	Basic + all SC deps. + blocks	850

TABLE III
EXECUTION ENVIRONMENTS.

VM Environment	RAM [GB]	Disk space [GB]	OS
Local VM	3	12	Ubuntu 12.04
Amazon EC2	1	8	Ubuntu 14.04
Google Cloud	3.5	10	Ubuntu 14.04

In the experiment, each of the four test workflows was deployed ten times. Figure 3 shows the average Execution Time (ET) needed to *deploy and enact* workflows, and the Standard Error of the Mean (SEM) as the error bars. The time includes provision of Docker containers, installation of the required dependency libraries, and deployment and execution of the tasks. ET was calculated as the average time starting from the submission of the blueprint until the completion of workflow execution.

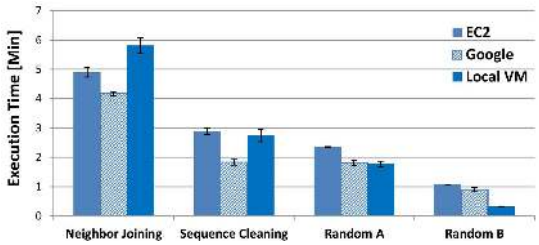

Fig. 3. Execution time for workflows enacted in different environments; the NJ workflow used the Basic and CentOS images, other three workflows used the Basic image only.

This experiment shows that our proposed approach is able to support workflow deployment on several Cloud platforms successfully. We used the same Service Template in each environment and the same scripts for all workflows. ET was significantly impacted by the structure, dependency libraries, and number of tasks in the workflow. In addition, the differences in the execution time for the same workflow deployed on different platforms was purely because of the variation in the time required to download the tasks and install different dependency libraries such as Java.

[7] https://github.com/WorkflowCenter-Repositories
[8] http://www.eubrazilcloudconnect.eu

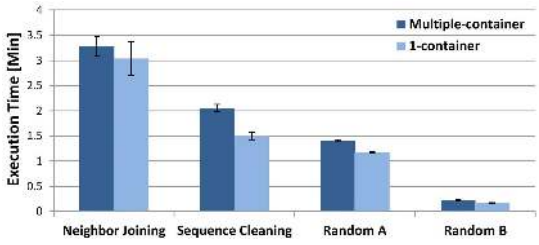

Fig. 4. Average execution time of single- and multi-container workflow deployments; all workflows used the Basic image.

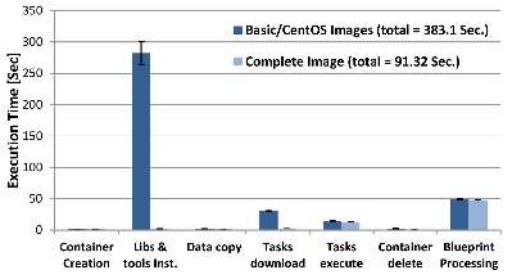

Fig. 5. Execution time for the steps in deployment the NJ workflow.

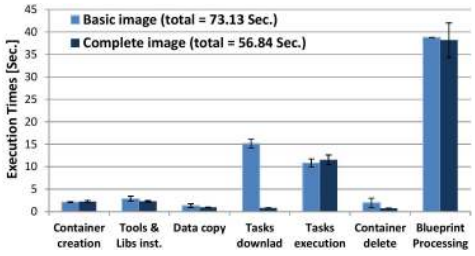

Fig. 6. Execution time for steps in deployment of the SC workflow.

C. Experiment 2: Single- and Multi-Container Deployments

With the ability to rapidly provision containers using Docker we wanted to investigate the overheads related to single- and multi-container workflow deployments. By using a separate container for each workflow task we can improve security and provide very good isolation properties for tasks. Therefore, understanding the related performance costs of such deployment strategies is important.

In this experiment we ran tests in two scenarios: (i) *multi-container* – each workflow task running in a separate container, (ii) *single-container* – one container used to run all tasks in the workflow. For both scenarios we used our local VM to deploy all four test workflows, and repeated each test 10 times. This time, however, we used only the *Basic* image to run tasks, thus ET included the provisioning of Docker container(s), installation of task specific dependency libraries and the actual execution time of all workflow tasks.

Fig. 4 presents the average execution time for the four workflows. As shown, there is little difference between the two scenarios: 14.9, 33.4, 13.9 and 3.2 seconds; or only about 2 seconds overhead per task. It reveals that the overhead of provisioning one container per task, which also involves the installation of dependency libraries, is not significant when compared to the deployment of the entire workflow in a single container where all tasks share the same container and most of the required libraries.

Again, the experiment shows variation in the execution time related to the network throughput. This time the execution of the SC, Random A and B workflows was faster than in previous experiment and the difference stems from the faster download time for task and library artifacts.

D. Experiment 3: The Influence of On-demand Deployment

In the previous experiment we showed the impact of the use of multiple Docker containers on the runtime of a workflow. Although the impact was relatively low, for all the workflows we noticed that the runtime was much higher than what we would expect running only workflow tasks. Therefore, we conducted an experiment to observe the influence of the on-demand installation and configuration of dependency libraries on the overall workflow runtime.

For this experiment we prepared two specialized images: *CompleteNJ* and *CompleteSC*, and used them to run the NJ and SC workflows in the multi-container mode. By using these specialized images we avoided the download and installation of any libraries and task dependencies during workflow execution. Fig. 5 shows that this part consumed over 280 seconds, the majority of the runtime of the NJ workflow if using only the *Basic* and *CentOS* images. Instead, when the task dependencies where preloaded in the *CompleteNJ* image, the installation time became negligible.

In Fig. 6 we show similar comparison for the SC workflow. In this case we used only the *Basic* image but none of the workflow tasks needed the time-consuming installation of the Wine library. The figure shows that the dominating part of the execution was the *blueprint processing* step calculated by subtracting from the total execution time the time taken by all tasks implemented by our lifecycle management scripts.

For the *Sequence Cleaning* workflow (8 tasks) the 'blueprint processing' time was about 38 seconds, and about 10 seconds shorter than for the other *Neighbor Joining* workflow (11 tasks). It shows the impact of the size of the workflow on the time required by Cloudify to process it.

It is important to note, however, that in our experiments the task execution times were relatively low. For longer-running tasks, the overheads introduced by our solution would play only a marginal role even if we use a generic image from the Docker Hub and decide to use the on-demand installation of the libraries. Although the on-demand deployment increases runtime of workflow execution, it reduces the burden related to image maintenance.

E. Experiment 4: Deployment with Different Docker Images

In most of the previous experiments we used the same image to deploy all tasks in a workflow. However, our framework is flexible enough to adopt other options to task and workflow deployment. The flexibility can help to address common

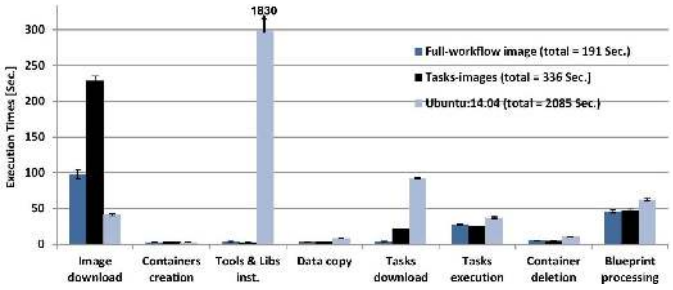

Fig. 7. Execution time of the *Neighbor Joining* workflow using three possible workflow deployment options.

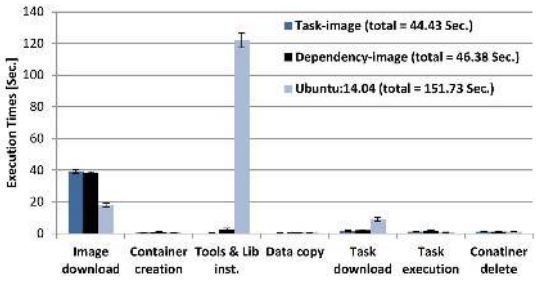

Fig. 8. Execution time of the CSVExport task deployed using three task deployment options.

challenges faced by the designers during the workflow development phase – frequent and irregular changes in task implementation. Therefore, in this experiment we investigate how various deployment options enabled by our framework can support workflow development. We look at them from two angles: the workflow and task level.

First, at the workflow level the designer can choose one of the three ways in which they may develop and deploy their workflow: using a pure-OS image, using a specific image for each task or using a workflow-specific image that encapsulates all workflow components and dependencies. We ran the *Neighbour Joining* workflow following these three ways: first one used the pure-OS Ubuntu:14.04 image from DockerHub, second one used seven task-specific images (note that the NJ workflow includes 11 tasks but some of them were instances of the same task type and so used the same image), third one used the *CompleteNJ* image with all dependencies and artifacts preinstalled. Fig. 7 shows the execution time in all three cases.

Clearly, the fastest execution was observed for the case which used the single, workflow-specific image. It was the fastest for most of the deployment steps and only the image download step ran noticeably longer than for the pure-OS case. That is because the *CompleteNJ* image is much bigger than the pure-OS Ubuntu:14.04 (c.f. Table II). On the other hand, it is smaller that the total size of the seven images required in the second case. The main drawback of the workflow-specific option is, however, increased effort needed for image maintenance. Every time a designer wants to update the code of any single workflow task they need to prepare a new workflow-specific image. Additionally, that option sacrifice isolation properties and is not available if any two workflow tasks have a conflicting set of dependencies.

At the other end of the execution performance was the slowest option which used the pure-OS image. It saved some runtime in the image download step as it needs only one, relatively small image for all the tasks but that was completely wiped out by very long time required to install on-demand all dependency libraries including Wine and Java (c.f. 'tools & libs inst.' in the figure, which took over half of hour). Despite having the longest execution time, this option may still be very useful during early stages of workflow development as it does not require any image maintenance. It is also particularly suitable for workflows built from scratch, in which case the designer may not yet realise whether there are any major costs related to dependency installation.

In between the two extremes is the option in which workflow tasks used specialized task images. It offers a good balance as the execution time is close to the fastest, workflow-specific case, yet it offers a good level of isolation and flexibility. The designer can combine tasks with conflicting dependencies and a change in one task requires update of only one, usually small image.

Looking at the same deployment options from the task level, the workflow designer has also a few options to choose from. First, they can decide to use the on-demand installation and embed a task that uses a pure-OS image. Second, they can prepare a Docker image that comprises the entire software stack needed by the task. Finally, they can decide to mix these two options and prepare an image with the software stack that includes all the dependencies, yet use on-demand installation for the task artifacts only. The last approach may be useful during the intensive task development phase when the developer frequently updates the task code while the core set of dependencies remains the same.

We prepared an experiment in which a workflow was configured with three tasks each realising different task deployment option. Fig. 8 depicts the task execution time for each deployment step. Again, there is clear trade-off between using a rigid approach with a specialized task image that gives the best performance, and the least efficient but most flexible approach which used the pure-OS image and relied on the on-demand installation of task and dependency artifacts. Yet, using the image with preinstalled dependencies only is an option that allows for flexibility required when task code changes frequently and which ran almost as fast the option that used a specific task image.

Importantly, the deployment options presented here can be mixed within a single workflow and also can change whilst the workflow and tasks undergo changes in their development phase. We expect that, initially, for a newly created workflow the designer would use specialized task images for the common, mature blocks such as I/O transfer because they rarely change. Whereas they would use on-demand installation for tasks specific to the workflow that are often created only for the purpose of a certain application. Then, once the

development of these tasks becomes less intense, the natural step is to build task specific images and focus on workflow design. Finally, at the point when the development phase of the workflow application becomes less intensive and/or the workflow is ready for the production use, the designer can capture a workflow-specific docker image with all tasks and dependencies preinstalled, which would offer the best performance for the users.

V. Conclusions and Future Work

In this paper we presented a new framework to build, deploy and enact scientific workflows. It integrates a TOSCA-based workflow definition with container-based virtualization. The most important benefits of using TOSCA to model workflows is the standard description language, improved portability and reuse of code. We used a small set of common lifecycle management scripts to deploy workflows and tasks irrespective of the workflow and Cloud platform they were running on. And by defining reusable Node Types for tasks and Service Templates for workflows, we enable new workflows to be built.

Using container-based virtualization, our framework can support execution isolation for heterogeneous workflow components and allows the underlying execution environment to be dynamically built and provisioned. Importantly, the combination of TOSCA and Docker adds greatly to the design-time flexibility. Given the low performance overheads related to container provisioning, designers can decide to run each task in complete isolation or in a shared container. Our framework allows task deployment to be easily split and merged across containers. Similarly, it enables image creation to be customised to best fit the actual implementation needs of task and workflow developers.

Overall, the proposed approach facilitates the reuse of task and workflow descriptions, and their artifacts, both at the level of the TOSCA definition and code distribution using Docker images. In the future we plan to investigate to what extent our approach can improve the reproducibility of scientific workflows and model a broader range of workflow structures including parallel enactment of sub-workflows.

Acknowledgments.: This work was partially supported by EU-funded project EUBrazil Cloud Connect, grant no. 614048 and by EPSRC grant no. EP/N01426X/1 in the UK.

References

[1] J. Wang, P. Korambath, I. Altintas, J. Davis, and D. Crawl, "Workflow as a Service in the cloud: Architecture and scheduling algorithms," pp. 546–556, 2014.

[2] B. Liu, B. Sotomayor, R. Madduri, K. Chard, and I. Foster, "Deploying Bioinformatics Workflows on Clouds with Galaxy and Globus Provision," *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pp. 1087–1095, Nov. 2012.

[3] C. a. Goble, J. Bhagat, S. Aleksejevs, D. Cruickshank, D. Michaelides, D. Newman, M. Borkum, S. Bechhofer, M. Roos, P. Li, and D. de Roure, "myExperiment: A repository and social network for the sharing of bioinformatics workflows," *Nucleic Acids Research*, vol. 38, no. May, pp. 677–682, 2010.

[4] Y. Zhao, Y. Li, I. Raicu, S. Lu, W. Tian, and H. Liu, "Enabling scalable scientific workflow management in the Cloud," *Future Generation Computer Systems*, no. 1, Nov. 2014.

[5] J. Zhao, J. M. Gomez-Perez, K. Belhajjame, G. Klyne, E. Garcia-Cuesta, A. Garrido, K. Hettne, M. Roos, D. De Roure, and C. Goble, "Why workflows break Understanding and combating decay in Taverna workflows," in *2012 IEEE 8th International Conference on E-Science*. IEEE, Oct. 2012, pp. 1–9.

[6] R. Qasha, J. Cala, and P. Watson, "Towards Automated Workflow Deployment in the Cloud Using TOSCA," in *2015 IEEE 8th International Conference on Cloud Computing*. IEEE, Jun. 2015, pp. 1037–1040.

[7] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 275–287, 2007.

[8] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, vol. 25482, pp. 171–172, 2015.

[9] OASIS, "Topology and Orchestration Specification for Cloud Applications version 1.0," pp. 1–114, 2013.

[10] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann, *TOSCA: Portable Automated Deployment and Management of Cloud Applications*, A. Bouguettaya, Q. Z. Sheng, and F. Daniel, Eds. New York, NY: Springer New York, 2014.

[11] A. Barker and J. V. Hemert, "Scientific workflow: a survey and research directions," *Parallel Processing and Applied Mathematics*, pp. 746–753, 2008.

[12] H. Hiden, S. Woodman, P. Watson, and J. Cala, "Developing cloud applications using the e-Science Central platform." *Philosophical Transactions of the Royal Society*, vol. 371, p. 20120085, 2013.

[13] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, and K. Wenger, "Pegasus, a workflow management system for science automation," *Future Generation Computer Systems*, vol. 46, pp. 17–35, 2014.

[14] J. Goecks, A. Nekrutenko, and J. Taylor, "Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences." *Genome biology*, vol. 11, p. R86, 2010.

[15] B. Balis, "HyperFlow: A model of computation, programming approach and enactment engine for complex distributed workflows," *Future Generation Computer Systems*, 2015.

[16] E. Afgan, N. Coraor, J. Chilton, D. Baker, J. Taylor, and T. G. Team, "Enabling cloud bursting for life sciences within galaxy," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 16, pp. 4330–4343, 2015.

[17] T. Binz, U. Breitenbucher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner, "OpenTOSCA A Runtime for TOSCA-based cloud applications." Springer-Verlag Berlin Heidelberg, 2013, pp. 692–695.

[18] J. Wettinger, U. Breitenbücher, O. Kopp, and F. Leymann, "Streamlining DevOps automation for Cloud applications using TOSCA as standardized metamodel," *Future Generation Computer Systems*, Aug. 2015.

[19] F. Li, M. Vogler, M. Claessens, and S. Dustdar, "Towards Automated IoT Application Deployment by a Cloud-Based Approach," in *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*. IEEE, Dec. 2013, pp. 61–68.

[20] G. Katsaros, M. Menzel, A. Lenk, R. Skipp, and J. Eberhardt, "Cloud application portability with TOSCA, Chef and Openstack," *2014 IEEE International Conference on Cloud Engineering*, pp. 295–302, Mar. 2014.

[21] G. Santiago, V. Andrikopoulos, R. Jim, F. Leymann, and J. Wettinger, "Dynamic Tailoring and Cloud-based Deployment of Containerized Service Middleware," in *IEEE 8th International Conference on Cloud Computing*, 2015.

[22] C. Pahl and B. Lee, "Containers and Clusters for Edge Cloud Architectures – A Technology Review," in *2015 3rd International Conference on Future Internet of Things and Cloud*. IEEE, Aug 2015, pp. 379–386.

[23] W. Gerlach, W. Tang, K. Keegan, T. Harrison, A. Wilke, J. Bischof, M. DSouza, S. Devoid, D. Murphy-Olson, N. Desai, and F. Meyer, "Skyport - Container-Based Execution Environment Management for Multi-cloud Scientific Workflows," in *2014 5th International Workshop on Data-Intensive Computing in the Clouds*. IEEE, nov 2014, pp. 25–32.

[24] J. Cała, E. Marei, Y. Xu, K. Takeda, and P. Missier, "Scalable and efficient whole-exome data processing using workflows on the cloud," *Future Generation Computer Systems*, 2016.