Newcastle University

School of Computing

Automatic Deployment and Reproducibility of Workflow on the Cloud using Container Virtualization

Rawaa Putros Polos Qasha

Submitted for the degree of Doctor of Philosophy in the School of Computing, Newcastle University

December 2017

Declaration

I declare that this thesis is my own work unless otherwise stated. No part of this thesis has previously been submitted for a degree or any other qualification at Newcastle University or any other institution.

Rawaa Qasha

I confirm that, to the best of my knowledge, this thesis is from the student's own work and has been submitted with my approval.

Supervisor

Prof. Paul Watson

Newcastle Upon Tyne, December 2017

Acknowledgements

First of all, I would like to express my deep sense of gratitude and profound respect to my supervisor Professor Paul Watson for giving me the opportunity to undertake a PhD and for his patient guidance, motivating encouragements and unreserved advice throughout my whole PhD study. He has supported me academically and morally from the beginning to the end of finishing this thesis. I have been extremely fortunate to have such a supervisor who cares so much about not only my research work, but also my personal career and issues.

My appreciation also extend to Dr. Jacek Cala who directly contributed to the discussion of my thesis work with great patience and immense care. He has been demonstrative and generous in his guidance and support and valuable advices throughout my whole PhD study.

I would also like to thank my examiners, Professor David Walton and Professor Alexander Romanovsky for the valuable viva discussion. I am also grateful for reviewing my thesis and also for providing constructive feedbacks and detailed comments.

I would like to thank my home country Iraq, the Ministry of Higher Education and Scientific Research in Iraq and Mosul University for providing the opportunity for me to undertake my doctoral study in Newcastle University.

Special thanks go to my sincerely friends especially: Dr. Emily Porter, Dr. Basim and his wife Ahlam, Dr. Ghanem and his wife Eman, Dr. Salim and Soulaf Kakel for their encouragement and support who directly or indirectly made this work possible.

I would also like to offer my gratitude towards many colleagues from School of Computing/ Newcastle University who provided me with a great deal of additional insight during the four years of study especially: Zhenyu Wen, Faris Llwaah, Hugo Firth, Dr. Matthew Forshow, Dr. Jannetta Steyn, Lesego Peter, Priyaa Thavasimani and Shaimaa Bajoudah.

Last but not least I would like to thank my parent for all support, love and patience

they have provided me over the years which was the greatest gift anyone has ever given me. Also I need to thank my sisters and brothers for their continuous support and love during the years of study.

Abstract

Cloud computing is a service-oriented approach to distributed computing that has many attractive features, including on-demand access to large compute resources. One type of cloud applications are scientific workflows, which are playing an increasingly important role in building applications from heterogeneous components. Workflows are increasingly used in science as a means to capture, share, and publish computational analysis. Clouds can offer a number of benefits to workflow systems, including the dynamic provisioning of the resources needed for computation and storage, which has the potential to dramatically increase the ability to quickly extract new results from the huge amounts of data now being collected.

However, there are increasing number of Cloud computing platforms, each with different functionality and interfaces. It therefore becomes increasingly challenging to define workflows in a portable way so that they can be run reliably on different clouds. As a consequence, workflow developers face the problem of deciding which Cloud to select and - more importantly for the long-term - how to avoid vendor lock-in.

A further issue that has arisen with workflows is that it is common for them to stop being executable a relatively short time after they were created. This can be due to the external resources required to execute a workflow - such as data and services - becoming unavailable. It can also be caused by changes in the execution environment on which the workflow depends, such as changes to a library causing an error when a workflow service is executed. This "workflow decay" issue is recognised as an impediment to the reuse of workflows and the reproducibility of their results. It is becoming a major problem, as the reproducibility of science is increasingly dependent on the reproducibility of scientific workflows.

In this thesis we presented new solutions to address these challenges. We propose a new approach to workflow modelling that offers a portable and re-usable description of the workflow using the TOSCA specification language. Our approach addresses portability by allowing workflow components to be systematically specified and automatically

deployed on a range of clouds, or in local computing environments, using container virtualisation techniques.

To address the issues of reproducibility and workflow decay, our modelling and deployment approach has also been integrated with source control and container management techniques to create a new framework that efficiently supports dynamic workflow deployment, (re-)execution and reproducibility.

To improve deployment performance, we extend the framework with number of new optimisation techniques, and evaluate their effect on a range of real and synthetic workflows.

Publications

Parts of the work within this thesis have been published in the following papers:

CONFERENCE

1. **R.Qasha**, J.Cala and P.Watson, *Dynamic Deployment of Scientific Workflows in the Cloud using Container Virtualization*, IEEE 8th International Conference on Cloud Computing Technology and Science, CloudCom December 2016.

2. **R.Qasha**, J.Cala and P.Watson, *A Framework for Scientific Workflow Reproducibility in the Cloud*, IEEE 12th International Conference on eScience, October 2016. The paper was selected as one of the four best papers of the conference.

3. **R.Qasha**, J.Cala and P.Watson, *Towards Automated Workflow Deployment in the Cloud Using TOSCA*, IEEE 8th International Conference on Cloud Computing, June 2015.

WORKSHOP

A part of this thesis has also been presented at the following workshop:

- **R.Qasha**, *Reproducibility of Scientific Workflow in the Cloud using Container Virtualization*, Docker Containers for Reproducible Research Workshop (C4RR), University of Cambridge, Cambridge 2017.

Contents

List of Figures

List of Tables

1

Introduction

Introduction

In recent years, Cloud computing has provided a new way to offer applications, software platforms, and computer infrastructure as services. Because organisations can now acquire virtualised computing resources on-demand, the adoption of Platform as a Service (PaaS) has supported rapid application deployment. This also enables systems to be scalable as the resources utilised by an application can rise and fall as needed to meet changing demand [80]. Consequently, the use of Cloud has been steadily rising, and an increasing number of enterprises are moving applications to the Cloud.

In parallel with the growth of clouds, scientific workflow has become an increasingly popular paradigm for scientists to formalize and structure the complex analysis that are now underpinning a growing proportion of scientific research. Here, scientific applications are structured as a directed graph of tasks that are executed to analyse input data [149]. The main advantages of workflows are that they can easily be assembled for a specific purpose from an existing set of available tasks [16] - even by novices - and that tasks can be re-used in multiple different workflows. These advantages are not limited just to the workflow's original developers, nor they are only specific to the research for which the workflow was generated; once created, a workflow can be shared, re-used and adapted by other scientists. The reasons for this are that workflows expose the structure of the computation, making it more understandable and re-usable by others. This can help other users: understand the experimental process, re-execute the workflow to replicate the original experimental results, apply the workflow to different data, or use the workflow as a starting point to design a new workflow for a different experiment. To realise this potential for sharing and re-usability, a number of public repositories such as myExperiment [53] and CrowdLabs [85] have been created to enable scientists to publish and discover workflows.

Unsurprisingly, as modern Cloud computing capabilities have emerged, workflow systems have been migrated to, or specially developed for, the Cloud. Clouds can offer a number of benefits to workflow systems, including the dynamic provisioning of the resources needed for computation and storage, which has the potential to dramatically increase the ability to quickly extract new results from the huge amounts of data

now being collected [66] [77]. Consequently, Cloud computing has become the main computing infrastructure underpinning scientific workflows. However, some significant problems remain. Tackling these challenges is the focus of this thesis.

The clouds provided by commercial entities and computing centres are offered to clients through diverse cloud management platforms with different functionality and APIs [59] [83]. It is therefore challenging to define workflows in a portable way so that they can be run reliably on different clouds. As a consequence, workflow developers face the problem of deciding which Cloud to select and - more importantly for the longer-term - how to avoid vendor lock-in. Cloud computing would be more valuable as an underpinning technology for science if workflow could be ported across different cloud environments. Without this, there is the danger that the workflow may become unusable if the supplier withdraws, or dramatically changes their offer. This would destroy the ability to reproduce scientific analyses performed by workflows, something that has become increasingly important to science in recent years.

Another major problem that affects sharing and reproducibility is workflow decay. A number of investigations have found that, a relatively short time after they were created, a large number of workflows either cannot be re-executed or do not produce the same results - this is known as workflow decay. Figure 1.1 presents the results of two studies, study1 [145] and study2 [87], that show the decay in two collections of workflows over a period of 5 (study1) and 8 years (study2).

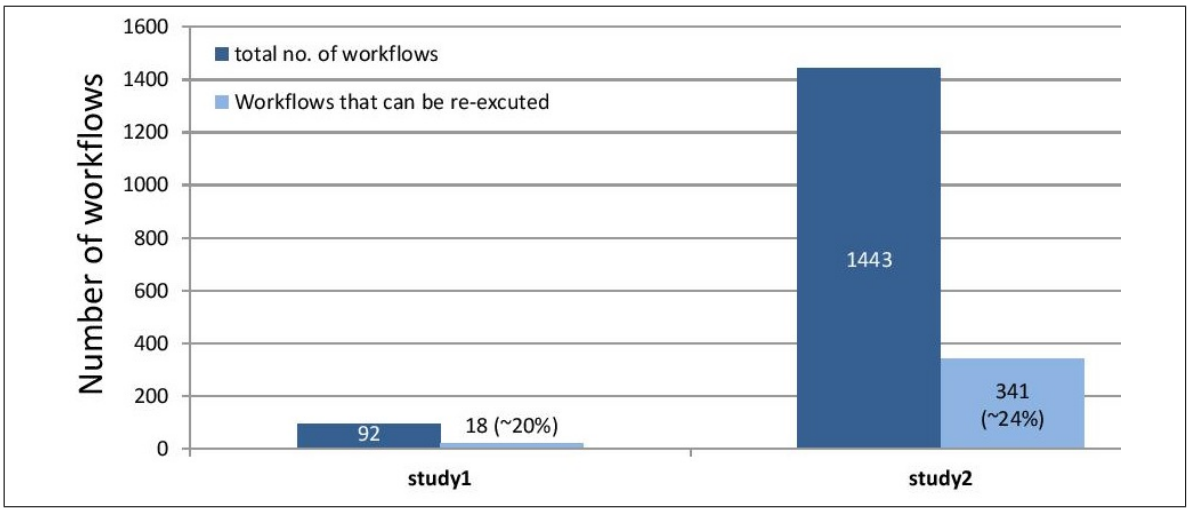

Figure 1.1: The results of workflow decay studies

The main reasons for the decay are:

- the resources (data and services) required to execute workflows are no longer available. For example, a service called by the workflow may have been removed by its owner.

- the resources (data and services) required to execute workflows are still available, but have been modified. For example, a service called by the workflow may have been upgraded by its owner so that it returns a different result. In the worst case, the interface to a service may have been changed, so that it can no longer handle the input(s) sent from its predecessor services in the workflow.

- changes in the execution environment on which the workflow (or one of its components) is dependent [18] prevent its successful execution. For example a change to a library, or the operating system, on which a task or the workflow manager depends, might prevent it from being executed.

Many approaches, technologies and tools have been developed to support the deployment of multi-service systems such as workflow applications. They attempt to reduce deployment complexity by providing descriptions of both the application components and their deployment target environments, abstracting the dependencies and automating the deployment process [122]. The main causes of deployment complexity are the heterogeneity of applications components and the target systems, the interactions between them and the constraints of execution policies. The heterogeneous services employed in composite workflows can have different dependencies and require different software stacks which may cause conflicts [23]. For example, different workflow's tasks may need the same library with different versions or each task might need to be executed on a specific version of the operating system.

Traditional deployment tools and technologies do not provide a complete solution for addressing these deployment complexities because of their limitation in resolving dependencies among components, and also due to a lack of support for heterogeneous application components and execution environments [130]. Moreover, most of these mechanisms have been developed for a specific type of application component, or for specific execution environments.

Therefore, we believe that a new deployment infrastructure tailored for workflow-based applications is required to tackle deployment complexities by:

(a) offering a way to comprehensively describe: application components and their requirements, target systems and their dependencies,

(b) providing a way to exploit this description to automate the deployment of these components onto a range of target systems (e.g. a range of different vendors's clouds),

(c) supporting the isolation of service execution to minimise disruption between services, isolation is required to protect the execution of the components from each other.

In this thesis we focus on addressing the problems of workflow portability and re-use through the design, implementation and evaluation of a novel system that meets the above design challenges. The approach significantly increases the ability to re-use and reproduce workflows, so reducing workflow decay. It does this by combining software modelling, automated deployment and virtualization.

Virtualization is at the heart of Cloud computing, as software encapsulated in VMs is deployed on cloud nodes. Recently, container-based virtualization techniques, such as Docker [1], have simplified and accelerated the deployment of applications in the Cloud [115]. When considering the case of a workflow application that consists of a set of heterogeneous components, each potentially relying on different dependencies, container-based virtualization offers the opportunity for rapidly creating efficient and isolated execution environments to build and deploy workflow components [74]. In this thesis we show how containers can be used to support the portability, dynamic deployment and repeatability of scientific workflows in the Cloud.

Virtualization alone cannot solve all the problems. It makes it easy to repeat exactly the same workflow execution, but we will show that it is often not enough to reproduce the experiment, possibly modified by using different parameters or input data. To achieve this, we also need to include Software Modelling, which enables the creation of

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

a detailed description of the various levels of the workflow, so enabling the automated deployment of virtualised components.

1.1 Research Goals

The overall aim of the thesis is to design, implement and evaluate a system for workflow modelling that offers a portable and re-usable description, so enabling automatic deployment on a range of clouds. The system should efficiently support the re-execution and reproducibility of scientific workflows both in the Cloud and in local computing environments.

To achieve this aim, in this thesis we investigate and address the following research objectives:

Scientific workflow modeling How can a scientific workflow and its heterogeneous components be modeled? Modeling should provide a comprehensive description of the workflow, its services, all required dependencies and relationships between these components. A workflow description should enable portability, re-usability and automatic deployment in a range of clouds.

Automatic deployment of scientific workflow How can we automate the deployment of workflows and their related services in the Cloud to achieve reusability, reproducibility and execution isolation? The solution should support the portability of workflow across a range of Clouds, and offer the flexibility to dynamically provision workflow tasks.

Workflow re-usability, re-execution and reproducibility How can we design a system to support the logical and physical preservation of scientific workflows so as to facilitate repeatability, reusability and also improve reproducibility?

Effective deployment of scientific workflows How can we optimise the performance of workflow deployment?

1.2 Contributions

Considering these aims and objectives, the main contributions of this thesis concerns four area: workflow modeling, deployment, reproducibility and performance optimization.

(i) A survey of the state-of the-art in cloud application deployment, scientific workflow modeling, deployment and reproducibility in Cloud computing. Existing methods and tools are described and critically examined.

(ii) A new approach to model scientific workflows, based on the TOSCA specification language which combines simplicity and re-usability. The approach enables the modelling of the workflow together with its components at a level that is independent of the cloud(s) on which it is deployed. It also enables automatic deployment on a range of clouds.

(iii) Development and evaluation of a new approach for the automatic deployment of scientific workflow in the Cloud. Our TOSCA-based modelling approach is integrated with container virtualization to dynamically deploy a set of existing workflows in a range of different scenarios, and includes support for the isolation of heterogeneous workflow components.

(iv) A new framework is designed and implemented to support repeatability and improve the reproducibility of scientific workflow. The framework covers both logical and physical preservation of the workflows. The framework is unique in incorporating software repositories to manage versioning of source code and effectually tracking changes. It is fully integrated with the automatic deployment system.

(v) Designing, implementing and evaluating a number of optimization techniques that significantly improve the performance of the deployment of scientific workflows, and consequently the performance of the reproducibility framework. These techniques automate the sharing and re-use of ready-to-run workflows and tasks packaged as images. A new algorithm is developed to automatically name and

select compatible task images, and is integrated with a version control system. This allows the automation of image creation, caching and sharing.

1.3 Thesis Structure

Chapter 1 describes the motivation behind the research, and highlights the research problems and main contributions of the thesis.

Chapter 2 presents background material and a summary of work related to the research described in this thesis.

Chapter 3 proposes our TOSCA-based modelling approach for the comprehensive specification of scientific workflow. The approach enables the portable and easy to reuse definition of the components and lifecycle management of workflows.

Chapter 4 describes a new automated approach for building, dynamically deploying and enacting workflows. We demonstrate that our modelling approach using TOSCA can be used effectively to facilitate automatic deployment of the workflow. In this approach, TOSCA-based modelling is integrated with container virtualization to support dynamic deployment.

Chapter 5 explores a design and prototype implementation of our new framework that supports repeatability and reproducibility of scientific workflows in the Cloud. We demonstrate how to utilize our TOSCA-based modelling to support logical preservation of the workflows and leverage container management techniques to physically preserve them.

Chapter 6 highlights a number of issues affecting the effectiveness of workflow deployment and ultimately the performance of workflow reproducibility. In this chapter, we propose new optimization techniques that significantly improve the deployment of scientific workflows including automatic caching, sharing and reuse of workflows and their components.

Chapter 7 summarises and presents the overall conclusions drawn from the work presented in this thesis, and proposes a number of directions for further work in the area.

2

Literature review

Summary

This chapter presents the related work that motivated and underpinned the work presented in this thesis. It also provides an overview of the concepts and tools used to create the solutions we present. It starts by describing some of the background information concerning the overall topic, including a brief primer on Cloud computing, container-based virtualization, scientific workflow and application deployment. Our discussion spans different areas which are related to the main focus of this thesis including: modeling, deployment and the reproducibility of scientific workflow as well as the optimization of provisioning. At the same time, gaps in the state of the art research are highlighted, and we describe briefly how this thesis tackles these gaps.

2.1 Cloud Computing

Cloud computing is a novel computing paradigm that offers computing facilities as a service. It provides resources to store data and host services on a massive scale [47]. A specific feature of Cloud computing is that it allows the provision of on-demand resources and customized computing environments with a pay-as-you-go charging model. Therefore, applications are increasingly being moved to the Cloud to exploit the rich set of resources available there [147]. As a result, Cloud computing has emerged as a distributed computing platform that has attracted the interests of researchers and practitioners in various areas.

Currently, there is no single agreed definition of Cloud computing among the scientific communities [142]. However, it is commonly agreed that the term Cloud computing basically refers to data centres delivering applications, platforms and infrastructure (hardware and systems software) as services over the Internet [10]. The major advantages and characteristics of cloud services that drive its widespread adoption can be summarised as [131]: (1) resources offered as services that can be accessed over the Internet, (2) on-demand scalability - provisioning of resources as needed to support the scalability of applications, (3) better resource utilization - cloud platforms efficiently managing resource utilization so as to meet demand from sets of applications; and (4) cost saving - cloud users only pay for the resources they use.

Typically cloud services has been categorised in three models: [89]: (1) *Software as a Service (SaaS)*– in this model, end users access providers' applications running on a cloud infrastructure. The applications are accessible from various client devices through a web browser (e.g., web-based email), an "app" or an application programming interface (API). Examples of SaaS include Microsoft Office 365 and Dropbox. (2) *Platform as a Service (PaaS)*– the services provided in this model are a computing platform for provisioning and hosting user applications. The services include: operating systems, programming language execution environments, web servers and databases. The users do not manage or control the underlying cloud infrastructure but have control over the provisioned applications. Examples of PaaS include Google App Engine (GAE) [2] and Salesforce's Force.com [3]. (3) *Infrastructure as a Service (IaaS)*, in this model, cloud providers offer fundamental computing resources such as virtual machines, storage, networks, and others. Therefore, users can deploy and run software, which can include operating systems and applications, using the provided resources. Example of IaaS are Amazon EC2 [1] and Microsoft Azure [76]. In this way, Cloud computing provides a new way to access and utilise IT resources including computing, storage, applications and networks.

When a Cloud is made available for open, public use, it is referred to as a "Public Cloud". In contrast, a "Private Cloud" refers to internal data centres used by a single organization or a third party, but not made publicly available; several open source cloud platforms, such as OpenNebula [93], Eucalyptus [97], and OpenStack [117], have become available to support accessing and using cloud resources and services on private clouds. The third model is "Hybrid cloud" (Federated Cloud) which is where an application uses a combination of private and public clouds, for example restricting confidential (e.g. medical) data to the private cloud [9].

Various challenges and obstacles have arisen with the growth of Cloud computing including vendor lock-in, service availability, data confidentiality/auditability, performance unpredictability and data transfer bottlenecks [10].

This thesis focusses on addressing one of these challenges - how to avoid vendor lock-in caused by a lack of common standards in the tools and APIs used by public clouds. We present a new system that allows workflows to be portable across a range of clouds.

This integrates the TOSCA specification language and container-based virtualization to build a reusable and portable description of a workflow which can be automatically deployed and enacted on different cloud environments. Therefore, this literature review focusses on work in the relevant areas: container-based virtualization, workflow modeling, deployment and reproducibility.

2.2 Container-Based Virtualization

Virtualization is considered one of the key enablers for Cloud computing - different kinds of computing resources such as storage, servers, and networks are virtualized to provide access to, and improve utilization of, the underlying physical resources [28] [74]. Virtualization has brought several benefits, such as reducing the number of physical machines by server consolidation and isolating the execution environments of applications. Using virtualization as a base, cloud technologies introduce the concept of computing as a utility, where on-demand computational resources are provided with the illusion of unlimited supply [32].

A Virtual Machine (VM) based infrastructure has been adopted widely in Cloud computing environments for elastic resource provisioning. It allows the resources of a single physical machine to be shared by multiple VMs for maximum efficiency, and gives benefits in terms of reliability and scalability. Each virtual machine (VM) [56] runs a separate operating system, independent of the operating system on the hosted machine. A hypervisor, running on each hosted machine provides virtualized computing resources (CPU, memory, disk, etc.) to each virtual machine. The VM-based approach supports the building of customised application stacks (including the operating system, middleware, and application components) as well as moving them between physical machines. The mobility of VMs is achieved by packaging them as VM images files that are portable because a hypervisor on the other host can instantiate a running VM from such an image. VM image files can also be stored in a repository for later usage. Popular examples of hypervisor-based virtualization are Xen [15] and VMware [135]. Today, hypervisor-based virtualization is used by cloud providers' to offer virtual machines for a wide variety of operating systems. It also achieves workload isolation as the components of different applications can be hosted on different VMs, so ensuring secure execution and minimal interference between them.

VM-based virtualization is not the only way of providing virtualized environments. In recent years, container-based virtualization techniques have emerged as a lightweight alternative [118]. As shown in fig 2.1, with hypervisors based systems, all required hardware components are virtualized, so as to allow the VM to run a fully independent

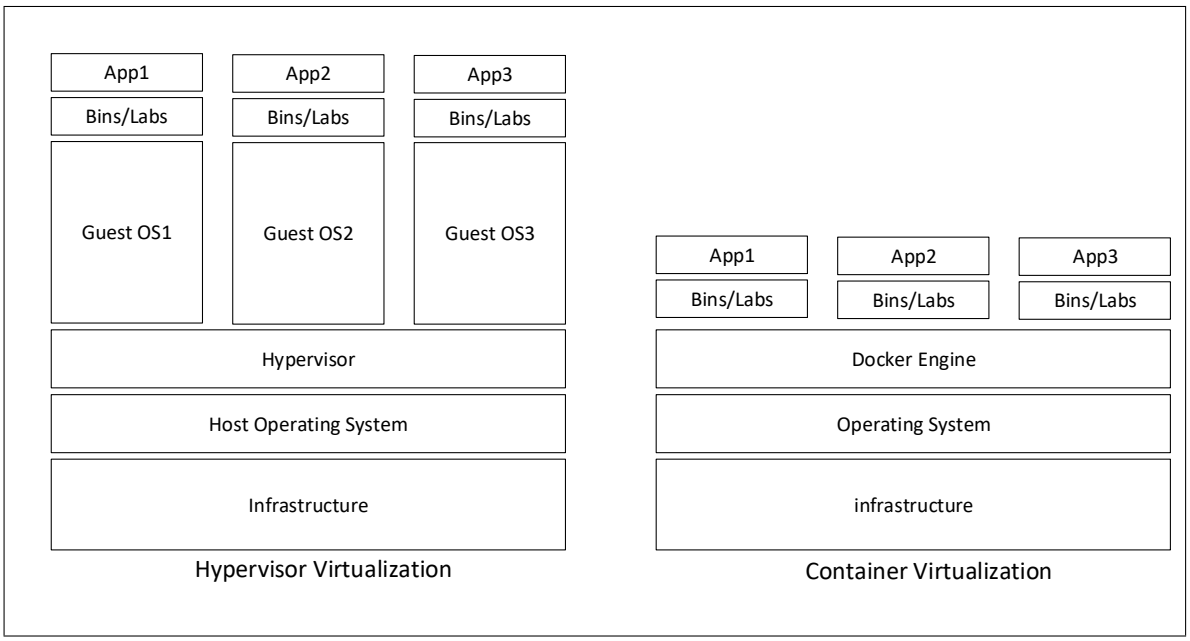

Figure 2.1: Hypervisor vs Container Virtualization.

guest operating system, whereas in container virtualization the virtualized resources share the kernel of the same operating system .

Container-based virtualization is not new [132], but recently there has been great uptake of Docker[33], an open-source implementation of operating system-level virtualization. Docker is based on open standards and runs on all major Linux distributions and on Microsoft Windows. As a result, it has established a strong and open ecosystem that several cloud providers now support.

Each Docker container is a running instance of a Docker image, where the image is a lightweight, executable package of the software stack that includes everything needed to run it: code, runtime, system tools and libraries as well as their configuration. Most Docker images are distributed via the Docker Hub , which is maintained by the Docker team. These containers share the kernel of the host system, but they each run in an isolated user space, and have their own file system and virtual network interfaces. Consequently, one container is unable to access others in an uncontrolled way. Further, techniques for insuring performance isolation prevent a single container from consuming all available computing resources [115].

The conceptual differences between hypervisor-based and container-based virtualization have an impact on the performance [74]. In general, the overhead of container-

based virtualization is lower because there is no separate guest operating system for each container this consequently produces a lightweight virtual environment. In addition, there is no need for hardware virtualization because the OS kernel is shared among the containers [116]. This, along with the portability of images across, leads to better utilization of available resources when compared with VM-based system. A comparative study [112] showed that a Docker container consumes only 6% and 16% of CPU and memory respectively compared with a corresponding VM. Test also show that processes running in Docler containers have almost a similar performance when running on physical machine. Docker image size is typically in the MBs range, whereas VM images are in the range of GBs - the reason is that the VM Image contains a full operating system whereas Docker images do not. This is a factor in why Docker images require only 60% of the corresponding VM time for booting and 2% for rebooting. This combination of advantages mean that overall, container-based virtualization tends to perform better than VM based virtualization, both when container is running, and in terms of the time needed to create and destroy containers [45].

For these reasons, cloud providers such as AWS Beanstalk5, Microsoft Azure7 and Google App Engine6 have started supporting Docker and encouraging this type of virtualization, as well as providing an ecosystem of services for deploying containerized components and applications [91].

In summary, Docker makes it fast and easy to build containers and to deploy them just about anywhere: including in private or public clouds.This makes them a suitable building block for the deployment of service-based architectures [46]. Therefore, one of the main goals of our work was to device an efficient way to dynamically deploy a distributed workflow application using Docker's container-technology.

2.3 Scientific Workflow

The workflow can be defined as a collection of distributed services and/tasks organised to accomplish a specific function or process [51]. Workflow technology facilitates the process automation and coordination of these distributed services and tasks by providing methodologies and software to model, configure and execute the workflow. Workflows have been used to capture information about the services/tasks at a level that describes their requirements, functionality and coordination. Originally, workflow was first adopted by the business community, known as business workflow, to provide glue for the distributed services and applied to automate repetitive tasks in business. During recent decade, the concept of workflow has been applied to capture and automate large-scale of science experiments, emerging the notion scientific workflow [39].

Scientific workflow is a well-defined, and possibly repeatable, pattern activities designed to capture a series of analytical processes which describe the design steps of computational experiments and to achieve a certain transformation of data between those steps [16]. The design and execution of many scientific applications require tools and high-level mechanisms. Therefore, many efforts have been devoted towards the development of management systems for scientific workflow that provide an environment to automate the scientific discovery process through the combination of scientific data management, analysis, simulation, and visualisation [123].

Scientific workflow can be considered as a model defining the structure of computational and data processing tasks. It should provide a declarative way to specify the high-level logic of the scientific experiment, be able to integrate existing tasks; services and data sets in different compositions that implement the architecture design of the experiment processes. The specification of workflow should be able to model the stream of data from one processing unit to another. In addition, an important characteristic of scientific workflow is the reusability: the ability to share for potential modifications and re-execution. The work presented in this thesis concentrates on addressing the challenges for scientific workflow management including: modeling, deployment and reproducibility [16].

Workflow has the advantage of making visible the dependencies necessary for the

management of a scientific process. As well as enabling software pipelines that perform multiple complex computaions. They can facilitate access to remote services, databases and distributed computers [123].

Over the years, scientific workflows have proved their effectiveness as an effective paradigm for programming complex scientific computations run on supercomputers or distributed computing infrastructures such as Grids, peer-to-peer systems and Clouds. Therefore, researchers in different disciplines have used them to orchestrate large-scale computations in a wide variety of scientific domains, including astronomy, bioinformatics, earthquake science, and physics [146] [63].

A workflow is composed of a set of tasks/services that are connected together to express data and/or control dependencies. Hence, each task/service represents a piece of work that forms one logical step in the overall process. There are a number of programming structures that have been adopted to represent the workflows. The most common pattern is the direct acyclic graph(DAG) which consists of set of nodes and edges [125]. The nodes represent tasks/services to be executed while the edges represent dependencies between the tasks. Two types of dependencies can be defined: 1) *data dependencies*, where the output data generated by a task is the input to another, and 2) *control dependencies*, where a task can only be started after the completion of other task(s)[40].

The usage of workflow technologies offers a number of important benefits over traditional approaches to structuring scientific computaions from a set of services (such as shell scripting) [40][43] including: (1) Simplifying the description of a computation: workflows provide a declarative way for specifying the high-level logic of the scientific process, hiding the low-level details that are not fundamental for application design (such as how data travles from one service to another). (2) Re-usability: the declarative nature of the workflow specification increases the ability of others to reuse portions of an existing workflow, or to modify it to meet new requirements. Therefore, scientific workflows can play an important role in the sharing, interchange, and reuse of scientific methods. (3) Failure management: each step in a workflow can be automatically rerun if a failure occurs (providing that it is idempotent). (4) Parallel and distributed processing: workflows expose the opportunity to run computations in parallel on dis-

tributed resources. (5) Provenance tracking: throughout the enactment stage of a workflow, information about the processed data, the executed computations, and the utilised resources can be captured and used later for validation of the method, and to increase re-usability.

The life-cycle phases of workflows including modeling, deployment and enactment are supported by a Workflow Management System (WFMS). We will now discuss these in turn.

2.3.1 Workflow Modeling

A large number of specifications, standards and frameworks have been created to model and manage workflows.

Most of the existing workflow models can be categorized into two classes [96]:

- Text-based systems: the workflow is specified in a textual programming language that can be composed by using a plain text editor or a script in a high-level language such as Python, Ruby or Java is used to generate the workflow specification. Business Process Execution Language (BPEL) is the most commonly used script-based approach to describe business workflows, and has also been used for science [127].
- Graphically-based systems: the workflow is specified using a number of simple graphical elements that correspond to workflow components such as nodes and edges. Most of the WFMSs provide a graphical tool for composing workflows [40].

There are number of formalisms and specifications used to describe general workflows, including the Unified Modeling Language UML [104], XML-based languages [11], DAGMan [61], Petri nets [129], DAX [41] and JSON. For specialised workflow applicarions, BPEL is a business focused standard, while a number of scientific workflow systems have been developed. Three of the main ones are Pegasus, Taverna and Hyperflow.

Pegasus WFMS [42] uses DAX for workflow description while Taverna [102] uses an XML-based language - the Simple Conceptual Unified Flow Language SCUFL. SCUFL

is a high-level language in which a workflow is defined as a set of local or remote services interconnected using data links [16]. HyperFlow [12] uses a JSON-based format to specify workflow structure including complex patterns such as parallel processing and control flow patterns.

Recently, an effort to design a common language for scientific workflows (CWL) [7] has been started, yet it is at the early stage and not mature enough for practical applications. CWL is a specification for describing analysis workflows and tools in a portable way across a variety of software and hardware environments that support the CWL standard. In addition, tools and workflows described using CWL can leverage Docker technologies. Despite its intention to provide a generic description for workflow processes and their dependencies, CWL does not include the ability to describe 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.

2.4 Application Deployment

Software (application) deployment is a process consisting of a number of inter-related activities including the release of software at the end of the development cycle; the configuration of the software, the installation of software into the execution environment, the activation of the software, and the configuration of the infrastructure for communication, isolation, and security [38][122].

Most of today's systems are distributed applications composed of multiple, diverse and related components. Therefore, different technologies and tools have been developed to address the deployment of a set of interrelated software components into heterogeneous environments by offering a description of the environments, an abstraction of the dependencies, and by automating the deployment process [64].

A number of studies have classified deployment approaches based on different criteria. Talwar et al [124] classified deployment approaches as script-based, language-based, and model-driven. Scripts were the first attempt to automate deployment and are a very good solution when there are small number of components. A better level of reuse can be achieved using object-oriented languages, because they can benefit from mechanisms such as inheritance and composition. Model-driven deployment brings new possibilities, as resources, systems, and applications can be separately modelled and hence more specific and intelligent tools can be developed. Moreover, models provide an abstraction layer over the heterogeneity of the managed system [111][138].

In contrast, in [139] the authors classified the deployment approaches as *Application-oriented* or *Middleware-oriented* deployment. In application-oriented approaches, the deployment process is specific to a particular application including its own components and any required middleware. The middleware oriented approach supports the deployment of middleware components that are not specific to a particular application. The later approach is more general and enables deployment re-usability for different applications using the same middleware.

The exploitation of cloud technology has enabled the full automation of application deployment so as to enhance the efficiency of applications management. The DevOps community provides various tools and approaches to automate deployment, with a

focus on on the deployment of a predefined application stack.

There are many deployment approaches, and they cover different aspects of the deployment process in various ways ranging from standards, tools, and systems targeting different infrastructures (domains). Each of them has its own features, advantages and disadvantages which are now considered.

2.4.1 Deployment Specifications

Various specifications attempt to standardize the mechanisms by which applications can be deployed on diverse target systems. These include: W3C-IUDD [134], UML, CIM [44], OMG DnC [103] and OASIS SDD [98] and TOSCA [100].

The OMG DnC specification is one of the most complete approaches. It provides comprehensive frameworks for development, packaging, and deployment of a range of component middleware [106].

The specification provides standard interchange formats for metadata used throughout the development lifecycle of component-based application, as well as runtime interfaces that are used for packaging and planning. These runtime interfaces use a component deployment plan to deliver deployment instructions to the middleware deployment infrastructure, which contains the complete set of information for the deployment and configuration processes for component instances and their related connection information [95].

However, there are number of drawbacks regarding specification size and complexities in the processing of deployment metadata in XML format. The DnC specification contains a big data model to describe the applications components and a deployment plan that grows substantially with increases in the number of component instances. Moreover, the deployment plan metadata defined by the DnC specification is not appropriate for fully capturing deployment ordering or dependencies [38][105].

OASIS offers two different specification. The first, aimed at lifecycle management in multi-platform environment, defines an XML-based schema called Solution Deployment Descriptors (SDD) for describing the installation characteristics of software [98]. SDD defines schema for two XML documents: Package Descriptors and Deployment

Descriptors. The Package Descriptors define the characteristics of the package that used for deploying the software. Deployment Descriptors define the characteristics of the software package, including creation requirements, configuration, and maintenance.

There are number of limitations on SDDs. In particular, it does not provide sufficient information to support deployment of the software it describes. In addition, it does not address how the artifact and their associated information are offered to the target environment in order to be managed [92].

A recent specification from OASIS is the Topology and Orchestration Specification for Cloud Applications (TOSCA) [75]. This offers a new approach to enable automatic deployment of multiple (cloud) applications and improve the portability of these applications in the face of the growing diversity in cloud environments. TOSCA provides two versions based on the XML [120] and YAML [101] modeling languages to define, build, and package the application building blocks in a completely self-contained manner. This allows their reuse by different applications in a standardized way [20].

The specification aims to improve the portability of multi-cloud applications by enabling an interoperable description of: application and infrastructure cloud services, the relationships between service parts, and the services operational behaviour, independently of the service creating supplier and any particular cloud provider or hosting technology [24].

The application of TOSCA is still in its early stages. Wettinger et al. [140] presented an integrated, standards-based modeling and runtime framework by using TOSCA as a uniform metamodel to combine a variety of DevOps artifacts (e.g. Chef cookbook[1], Juju Charm [2]) and enable them to interoperate seamlessly.

In [75], the authors propose to use TOSCA to specify the components and configuration of Internet of Things (IoT) applications. Kostoska et al. [72] presented an implementation of TOSCA to enable a custom University Management System to be deployed in a flexible and portable manner. A proof of concept (PoC) project to investigate the current state of the art in the portability of cloud applications is described in [67]: a multi-tier web application was modeled according to TOSCA and

[1]https://www.chef.io
[2]https://juju.ubuntu.com/docs

TOSCA2Chef execution environment was developed to allow he deployment of the application in a Cloud. None of these previous efforts have, however, tried to use TOSCA for scientific workflow enactment.

Therefore, in this thesis we explore the use of TOSCA to define scientific workflows in a comprehensive and portable way, and also design and build a system for their automatic deployment.

2.4.2 Deployment Tools

In addition to the above specifications, a wide set of other approaches and tools have been proposed and developed to achieve automatic deployment with various capabilities and support. These provide either a stand-alone software, command-line interface, or a GUI web application such as HTCondor [126], Wrangler [65], Juju and Docker.

HTCondor is an open-source High Throughput Computing (HTC) management software framework for enormous group of distributed environments. It comprises a set of software tools which implement and deploy high throughput computing on distributed computers. When the users submit their serial or parallel jobs, HTCondor performs the following tasks: places the jobs in a queue, selects when and where these jobs can be run based upon a policy, controls their progress, and finally informs the user upon completion [126]. Although HTCondor can find environments for the execution of programs, however it unable to schedule the running programs based on their required dependencies [123].

Juve and Deelman [65] presented Wrangler, a system that can be used to deploy distributed applications automatically on cloud infrastructure by provisioning, configuring, and managing virtual machine. In Wrangler users describe the desired deployment to a coordinator which is a web service responsible for virtual machine provisioning. The co-ordinator uses an agent in each machine to install and configure the software and services. The system can deploy applications across a range of clouds using VM images, enables users to define custom behaviour of their applications using plugins, and allows the specification of dependencies between provisioned VMs. But all the VM images used in the provisioning process required to be prepared manually in a prior step.

Ubuntu presents an open source project called Juju that offers a service orchestration framework to deploy and manage applications independently of the underlying hardware using components called 'Charms' on public, private or hybrid cloud. Juju charms define the applications as services, each charm is a structured packages of files containing metadata, configuration data, and some extra support files. Charms encapsulate information on how the services can be deployed and configured properly on a Cloud. They can be fetched from an external charm store, stored in a local repository, or written by the user. Juju supports the deployment of applications using Charms to AWS EC2, OpenStack, Azure, and even user own Ubuntu based laptop. It provides a command-line interface and web application which can be used for designing, building, configuring, deploying and managing the infrastructure [17]. However, Charms are Linux oriented, therefore the portability of Juju applications is limiting .

As described in section 2.2, one of the latest tools for application deployment is Docker. Docker provides mechanisms that enables packaging of applications in a way that facilitates their deployment and execution [74] by wrapping an application in a container with all the above-kernel software required for execution. The containers can be created either manually or automatically. It has rapidly achieved widespread use for deploying multi-service applications, because of its lightweight software stack packaging, and its support for execution isolation. In addition, the portability and self-sufficiency of containers provides the ability to run them on a variety of different hosts. Docker is now used as a building block for deploying web apps, database services and service-based architectures [46].

However, all these existing deployment mechanisms have some constraints and are not fully able to remove the complexity of deploying a distributed system. Traditional mechanisms for deployment including tools and technologies, do not have the capability to manage the complexities of service deployment because of the restrictions they place in resolving dependencies among components, and a lack of support for heterogeneous environments.

2.5 Workflow Deployment

As described earlier, scientific workflows are considered a key mechanism for representing and managing distributed and complex scientific applications. Therefore, to manage and deploy a workflow as a type of distributed application, a software platform needs to provide integration and coordination of its components, services and tools [81].

The management of workflows are supported by Workflow Management Systems (WFMS) which are software platforms that provide functionality such as workflow modeling, execution and monitoring [148]. The key function of a WFMS during the workflow execution (or termed enactment) is coordinating the operations of the individual activities that constitute the workflow. Different scientific workflow management systems have been developed that allow scientists to combine services, tools and infrastructure for their research. Examples of well-established systems are Pegasus, Galaxy [54], e-Science Central(e-SC) [58], Taverna, Kepler [82] and HyperFlwo [12].

Most of the scientific Workflow Management Systems (WfMS) focus on workflow expressiveness and ease of modeling. Only a few solutions such as e-SC, Pegasus, Galaxy and, more recently, HyperFlow tackle the problem of deployment of scientific workflows in a distributed environment.

Scientific workflow systems have formerly been applied over a number of execution environments such as workstations, clusters/grids, and supercomputers. Cloud computing with its huge, on-demand resources has great potential for deploying and running scientific workflows [80] and this is discussed in the next section.

2.5.1 Deployment of Cloud Workflow Systems

With the emerging of Cloud computing, the trend is for distributed WFMS to migrate to the Cloud [64]. Migrating and running scientific workflows on the Cloud has the potential to provide multiple benefits [80]: 1) Scalability, as cloud platforms can offer a huge amount of computing resources and storage space to enable the flexible scaling of the scientific problem addressed by the workflow. 2) Deployment flexibility through

exploiting virtualization technology on a cloud platform, where different workflow environments can be either preloaded using VM images, or dynamically deployed in VM instances. 3) Improving Resource utilization with the on-demand resource allocation mechanisms offered by the cloud. 4) Flexibility in the trade-off between performance and cost by using different clouds (private, public or hybrid).

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. Therefore, potential advantage 4 above is difficult to realize.

Today there are number of WFMS that were based on Grid computing such as Taverna, Gridbus [26], Triana [31], Kepler, and others. There are however few WFMS on the Cloud, we now discuss those that do exist.

e-ScienceCentral (e-SC) [3] is a cloud-based workflow management system that provides the capability 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 (via an API). The system implements a simple dataflow model in which a workflow comprises a set of interconnected blocks. Links between blocks denote data dependencies, while data between blocks are passed as files in the engine's local file system.

e-SCworkflow blocks can be of different types (Java, R, Octave, etc.) and the definition of a block also contains software dependencies that must be met. Before running a block, any currently unavailable libraries are downloaded from the server on demand. Only once all software dependencies are met, can the engine start executing the block. This gives the e-SCworkflow engine a way to deploy blocks and libraries, and thus makes the engine generic and independent of the workflows it is to enact.

Pegasus is a well-established WfMS [43]. Pegasus users define workflows as abstract and resource-independent and they are then mapped by the system into concrete, platform-specific execution plans. The plans are enacted by HTCondor DAGMan which tracks dependencies and releases tasks as they become ready to run, whilst

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

HTCondor Schedd runs them on available resources. Importantly, Pegasus uses a *Transformation Catalog* to find user executable files that implement workflow tasks. 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, whilst the Transformation Catalog supports only discovery of user executables.

One of the widely used scientific WFMS is Galaxy. Galaxy is an open source, web-portal platform designed to address the problems of scientific tools accessibility, reproducibility and transparency. It is capable of connecting bioinformatics tools in pipelines. Galaxy offers the ability to: share workflows and data, and to extend researcher tools with the help of system deployers. Liu et al [77] extended the existing Galaxy workflow system using a number of tools to add data management capabilities, domain-specific analysis tools, automatic deployment on the Cloud, a cloud provisioning tool, and support for validating the correctness of workflows. The automation of Galaxy deployment on the Cloud is achieved using the Globus Provision-based method [78] which is a tool for deploying a distributed computing system automatically. Globus Provision requires a topology description file to deploy a Galaxy instance on Amazon EC2. HtCondor is used to run specified Galaxy jobs on remote clusters. Much like most other WFMS, Galaxy relies on external tools and datasets, and Galaxy ToolShed and Data Managers are solutions that facilitate the installation of the desired tools and datasets in a specific Galaxy instance [6][5].

Crucially however, the execution of workflows is considered separately from the installation of dependencies, making workflows usable only if all the dependencies are available prior to execution. The Galaxy team have made an effort to alleviate these problems by offering a dedicated cloud data and VM images which are preconfigured with a suite of the most common tools and reference data.

Recently, HyperFlow [12] WFMS has been developed to provide a programming model and enactment system for scientific workflow. To support workflow programming, the system combines a declarative description of the structure of a workflow and a low-level implementation of workflow tasks using a scripting language. The system uses VM images with prepackaged user application tools. At run-time, the HyperFlow executor

is able to instantiate the images according to the configuration description submitted alongside the workflow definition. However, to create an executable workflow, a VM images containing the required software packages must be pre-prepared by the user [13].

Most of the existing WFMSs use a very specific workflow definition language which limits their portability and only few of them tackle the problem of scientific workflow deployment in a distributed environment that enables workflow components provisioning over different environments. Instead, in this thesis we present a method to define scientific workflows in a comprehensive and portable way using TOSCA which we integrated with Docker technology for achieving dynamic and automatic workflow deployment.

2.5.2 Workflow Deployment using Virtualization

As described above, virtualization technology has brought number of valuable advantages for Cloud computing. One way in which virtualization can be used to improve automatic deployment is where the application and its associated components are pre-installed with all of their dependencies as a "virtual appliance" in a virtual machine image. A virtual appliance provides a simple, unified tool for service deployment by encapsulating a complete custom environment. Components are deployed by instantiating virtual machines with their virtual appliances on different Infrastructure as a Service (IaaS) cloud systems or on physical machines [69].

There exist a number of works in the literature and software tools that address the deployment of different applications using virtual machines [122][68]. In [143] the Typical Virtual Appliances (TVAs), a template to generate virtual appliance for the frequently used services, is proposed for a Cloud computing data centre in order to address the problems of service management.

Zhang at el [144] exploit user-level virtualization technology to propose a framework for improving the deployment flexibility for Cloud computing by decoupling the application software from the VM. The proposed method is an enhancement to virtual appliances in order to achieve further application isolation from the OS.

The service oriented multiple-VM deployment system (SO-MVDS) [49] provides a

model for template management to create and configure virtual appliances containing on-demand services. Furthermore, the system includes a mechanism for deploying multiple services automatically and dynamically within virtual appliances.

More recent work in [27] presented a platform that offers the ability to provision on-demand virtual machines and automatically install software representing scientific activities and data dependencies.

Moreover, a number of WFMSs have adopted virtual machines for various deployment scenarios. Galaxy WFMS uses virtual machine images to deploy a personal Galaxy server locally or on a cloud infrastructure with particular tool sets in order to ease the burden of installing and administering tool dependencies. In addition, the CloudMan [4] application has been adopted by Galaxy to support execution on multiple clouds and offers an automatically and dynamically scalable virtual cluster with a preconfigured Galaxy application and data. The virtual cluster is used to execute the jobs in workflows [5].

In Pegasus, workflows can be deployed in the Cloud by configuring cloud VM instances as an HTCondor pool. A number of VM images are prebuilt and configured to contain HTCondor, the Pegasus client tools, and the application. Workflow jobs are distributed across virtual machine instances coordinated by HTCondor and then executed [43].

Unfortunately, solutions based on VM images demand additional effort by users because, over time, users will need to add new, or update existing, application tools. That requires effort to maintain and rebuild the images, which is rarely supported by the WFMS itself. Without this effort, workflow decay will occur.

2.5.2.1 Deployment with Container-based Virtualization

Although many current cloud platforms and deployment techniques are based on virtual machines, virtualization techniques based on containers have emerged recently as an alternative to hypervisor-based virtualization for addressing the deployment of various applications [118]. Further, the evolution of container-based virtualization techniques has a significant impact on simplifying and enhancing the development and deployment of distributed applications on the cloud. The key reasons for the adop-

tion of this technology in application deployment [45] and also in the Cloud [115] are: they are more lightweight than VMs, faster start up times, faster processing and lower storage overhead [128].

Recently, these techniques have been used to deploy different type of applications. Santiago et al. [115] proposed a dynamic tailoring and deployment process of service middleware components geared towards the cloud environment. Their work leveraged a container-based virtualization environment for enabling the assembly, provisioning, and execution of dynamically built instances to satisfy the service middleware communication requirements of specific applications.

The authors of [71] presented *Yard*, a Docker-based deployment system which can deploy web applications to different platforms using a number of isolated containers. In addition, the paper includes a comparison between the deployment of the same application using Docker and VMs, which shows that VM-based deployment requires more effort by the user.

In [52] the authors presented *Skyport*, an extension to their data analysis platform that adopted Docker containers for automated deployment of scientific software applications. They manually created a separate Docker image for each task that executes a tool and then used that image to deploy the workflow.

The work presented in [70] demonstrates the using of Docker technology as a part of a framework to deploy microservices on the Amazon EC2 Cloud and identifies important elements relevant to the performance of microservice platforms.

In [79] a container-based platform is presented to run scientific workflow using number of cluster systems to build the execution environments required to run Galaxy workflow. In this work, the containers used to host the workflow execution engine and scientific tools required to run the workflow must be created manually before workflow runs. In addition, all the required software and tools must be per-installed in order to run workflow tasks. In contrast, our deployment approach dynamically creates the containers during the deployment time and installs all the required tools and dependencies on-demand.

Most of the above approaches are not developed for workflow deployment and they need

manual creation of the Docker images required for the deployment of an application. In contrast, our approach is different to these approaches. It dynamically injects into a Docker container the full software stack required to execute a workflow task; this includes all dependency libraries, tools, and the task's code. In addition, workflow and task images are created automatically and dynamically during workflow execution.

2.6 Workflow Reproducibility

One of the potential advantages of WFMS is their ability to allow workflows to be shared and re-used, and to become a building blocks for new experiments [36]. The ability to re-use the workflow is explained by two related terms that have often been used interchangeably although they are distinct: 1) *repeatability* refers to the ability to repeat exactly the same workflow execution in the same computational setting and produce the same result, 2) *reproducibility* refers to the re-usability of the experiment with different parameters or input data, the major goal being to validate the same scientific conclusions [87]. However, scientists and users find that it is difficult if not impossible to repeat or reproduce workflows over time, due to the serious problem of workflow decay as discussed in chapter 1 [57].

In the recent years, a number of different methods have been devised to support re-usability and facilitate the building of new workflows. Galaxy WFMS enables sharing and publishing of workflows and related objects, such as datasets and tools via the web [5]. Taverna and Kepler workflows can be shared via myExperiment [53] which is a public repository allowing the sharing of workflows including computational descriptions and visualisations of their components. However, workflow sharing provided by WFMSs and public repositories is limited to offering workflow structure and description as artifacts that need manual preparation of the system for workflow re-execution.

A number of analyses and research efforts have already been conducted to determine the salient issues and challenges in workflow reproducibility and the main causes of workflow decay [145][8][14][48]. These issues can be summarized as: 1) insufficient documentation and non-portable description of a workflow including missing details of the intermediate processing tools, inputs, outputs and execution environment. 2) unavailable execution environments when the execution of workflow requires a specific operating system type or version with some particular configuration that may no longer available or has been changed. 3) missing third party resources: most of the workflows could not be re-executed because of missing or modified software on which dependencies required for their executions such as external web services. 4) missing data required to repeat the workflow execution.

The authors in [145] conducted an empirical study on a collection of Taverna workflows and presented an analysis showing that approximately 80% of the workflows used in the test either could not executed or produced different results (if testable). A more recent study [87] has been conducted on a set of 1500 workflows obtained from the myExperiment platform to quantify how many of workflows can be easily re-executed. The study shows that only 341 (about 24%) of workflows out of 1500 could be re-executed.

There have been various attempts proposed in the literature or software tools to address repeatability and reproducibility of scientific workflows, most of them follow one of the two directions: (1) describing a workflow and all its components, called logical preservation/conservation, or (2) packaging the components of a workflow, known as physical preservation/conservation.

2.6.1 Reproducibility with Logical Preservation

Logical preservation techniques (also referred to as specification-based) focus on supporting a detailed description of the workflow and its components with enough information for others to enable the reproducibility of a similar workflow in the future.

There have been various efforts and attempts proposed in the literature to capture the details required to reproduce scientific workflows. In addition, to promote workflow reusability and support its reproducibility, a number of scientific workflow repositories, as described earlier, have been built and launched such as myExperiment, CrowdLabs [85] and the repositories offered by some of the WFMSs.

In the past decades, one of the most common approaches for addressing the reproducibility of scientific workflow is by capturing the provenance information about the workflow results which can be considered another technique of logical preservation [110][88]. The provenance-based approaches address the conservation of data (input, intermediate and final results), processing units descriptions and data flow dependencies such as in the works presented in [141][14][94]. In addition, a number of WFMS provide provenance capturing as a means to support their workflow reproducibility such as eScienceCentral [94] [141] and Galaxy, while in [113], the authors

presented an approach to capture and share detailed information about the execution environment in which the computational experiments have been conducted. They use a set of semantic vocabularies to specify the resources involved in the execution of a workflow. However, other studies have shown that sharing only the specifications of a workflow is not enough to ensure successful reproducibility [18] when the necessary tools, dependencies and execution environments are no more available.

More recently, Hasham et al. [55] presented a framework that captures information about the cloud infrastructure used for workflow execution and interlinks it with the data provenance of the workflow. They propose to offer workflow reproducibility by re-provisioning a similar execution infrastructure using the cloud provenance and then re-execution of the workflow. Although the approach enables re-execution, it is unable to track and address changes to the original workflow.

Another approach that can be considered as logical preservation is presented in [18] which proposed Research Objects for the preservation of scientific workflows. Research Objects can aggregate various types of data to enhance workflow reproducibility including: workflow specifications, description of workflow components and provenance traces. However, they do not include enough technical details about dependencies and the workflow execution environment to easily allow re-enactment.

Provenance capturing enables the encapsulation of an exact trace of a past workflow execution, which can then help in its re-execution. Nevertheless, provenance usually describes only the abstract layer of a workflow because detailed traces of the use of execution environment (e.g. at the OS level) quickly become overwhelming. Further, the specification-based mechanisms provide various details that can help in understanding the workflow and its components. Yet, they are still insufficient when some of the required dependencies change or become unavailable, in which case the ability to reconstruct the same execution environment is lost.

2.6.2 Reproducibility with Physical Preservation

In the physical preservation approaches, the workflow is conserved by packaging all of its components to create an identical replica that can be reused later. Therefore,

a packaging tool is required to aggregate all workflow components as well as some essential resources such as tools and the software stack [113]. However, such packaging approaches demand considerable maintenance efforts, and there is no guarantee that it will not change [50]. Importantly, a fundamental problem with packaging approaches is that they are limited to workflow repeatability.

To implement the packaging of workflows, virtualization mechanisms have been used and number of packing tools have been developed. Chirigati et al. proposed ReproZip [35]. It tracks system calls during the execution of a workflow to capture the dependencies, data and configuration used at runtime, and to package them all together. Then the package can be used to re-execute the archived workflow invocation.

Other researchers have used either hypervisor-based or container-based virtualization mechanisms to achieve physical preservation of a workflow. Hypervisor-based has emerged as a promising mechanism for reproducing the results of scientific computation. In one such an approach, the entire experiment represented be a scientific workflow is conducted within a virtual machine and the state of a virtual machine is saved as an image (VMI) image so that the resulting image can be shared. Several papers have proposed utilizing these concepts and mechanisms to support workflow repeatability and improve its reproducibility [121][60].

The work presented in [62] introduces an approach that builds on the concept of virtual appliances to enable workflow repeatability. In this work a new resource abstraction is defined, called a workflow virtual appliance (WVA). The WVA is a virtual appliance that encapsulate all components required to deploy a workflow, including the software, data and its execution environment.

The main advantage in using VMIs is that they allow the complete experimental workflow and environment to be easily captured and shared with other scientists [119]. However, the resulting images are large in size and so costly for public distribution [22]. And despite the fact that the packaging mechanisms allowing workflows to be re-executed (i.e. allow repeatability), they usually do not convey a detailed and structured description of the entire computation, relevant dependencies and execution environments, which would help in understanding the package contents. Therefore, their ability to reproduce or even reuse a packaged workflow in other contexts (e.g.

using different input data, parameters or execution environments) is often limited.

Similarly to Virtual Machine hypervisors, Docker allows workflow applications along with all necessary dependencies to be encapsulated in a container image [34][21]. Docker provides many attractive features that can be exploited to support workflow repeatability: 1) it is easy to build Docker containers using simple scripts so they can be shared and re-used using Dockerfile. 2) there is minimal overhead in running a Docker container, and creating a Docker image. 3) the containers and images are lightweight compared with VMs and VM images.

However, even if these approaches can offer a convenient mechanism to preserve workflows, they still lack a structured description of the aggregated components. In addition, they are limited to packaged resources and dependencies, and lack the flexibility to change the components or dependencies in an already packaged workflow - this is necessary for tracking and handling changes so as to avoid workflow decay.

2.7 Optimization of Workflow Provisioning

As mentioned in the previous section, scientific workflows are not only useful for modeling and managing the computation, but also as a means of sharing experimental methods. Once shared, they can help scientists to understand the overall experiment, and may be used as an essential building block to build new experiments or to repeat the experiment and replicate the original results. If they are re-used repeatedly then deployment performance becomes important.

In general, deployment optimization might be defined as the process of finding the optimal placement of application components over the computational resources and effectively provisioning all components to minimize the total deployment time [25]. Most of the efforts carried out in the field of deployment optimization are related to making decisions concerning the best models for application deployment and deployment planning [73], i.e. finding the optimal placement of components over computational nodes from the performance perspective prior to the process of deployment. For example, the work presented in [25] concentrates on the optimization of components' distribution over the computational node to lower the deployment overall time using an analytic model to search and evaluate possible deployment scenarios at design time. In contrast, our approach focuses on the automatic optimizing of component provisioning at deployment time.

There are a few efforts addressing the challenges of optimizing the provisioning process of workflows and other distributed applications using different mechanisms.

Some of the WFMS support optimization for the workflow provisioning process using a caching technique. In the e-SC workflow management system, all the required blocks for workflow execution are downloaded during execution time and all unavailable libraries required for running the block are downloaded on demand from the server and cached in the execution environment [30]. Therefore, all downloaded blocks and dependencies are available for subsequent use to re-execute the same or new workflow. Similarly, in this thesis we propose to implement caching essential workflow components such as tasks artifacts and dependency packages in the execution environment. Moreover, we also provide multi-level caching of a deployable components (workflows

and tasks) to support provisioning optimization and facilitate the sharing and re-use of these components.

The authors in [133] introduced an optimization approach that can be integrated with their existing approach for automatic provisioning of services. They proposed a number of optimization strategies for reducing the cost and time for services that were used repeatedly. This approach tried to achieve optimization of service-based application deployment by provisioning the services once and keep them running for subsequent use. In the work presented in this thesis, we tried to apply an optimization strategy that is similar in some aspect, for the full provisioning of workflow tasks, including the execution environment and the required software dependencies and re-use them for many. We achieved this by packaging the task with full software stack using a Docker image which can be utilized by subsequent usages.

In [37], Czarnul presented a model and an integrated system for the definition, run-time optimization, and execution of workflows. The system addresses the optimization of workflow execution when service availability is limited and might change over time. Their solution is limited to workflow services; however, they applied a similar strategy to our approach. Prior to the running of a workflow, services are chosen statically based on the information available and during the execution time services are reselected if some selected services become unavailable or new ones appear. In contrast, in our approach, the Docker image used for provisioning a task is specified at the workflow modeling stage and during deployment time. Meanwhile, a compatible task image may be created by other users, and if so this can be selected and used to reduce provisioning time

Beside the performance criteria, number of researches have considered best resources utilization and cost optimization for the workflow management in the Cloud [137] [150] [84]. The authors in [137] presented a new algorithm to deploy workflow applications on federated clouds that addresses application security requirements and optimises the deployment in terms of its entropy and monetary cost, taking into account the cost of computing power, data storage and inter-cloud communication. In [84], the authors developed a new algorithms for efficient management of workflow ensembles concerning budget and deadline constrains on the IaaS clouds. While in our work

we only concern about performance optimization of workflow deployment in term of minimizing the deployment time. We may consider other optimization criteria for the future work direction.

Our work presented in this thesis varies from the available approaches in the literature for workflow sharing in that we provide sharing ready-to-execute workflows/tasks. Further, our approach supports optimization of workflow provisioning by implementing multi-level caching for various workflow components and automates the process of selecting an appropriate image for provisioning workflow/task.

3

TOSCA-based Modeling for Automated Workflow Deployment in the Cloud

Summary

This chapter presents a new approach to modeling scientific workflows in a portable and reusable way. We show how TOSCA, a new standard for cloud service management, can be used to systematically specify the components and life-cycle management of scientific workflows by mapping all the elements of a workflow onto entities specified by TOSCA. Later in this thesis we demonstrate how this enables the definition of workflows that are portable across clouds, resulting in greater reusability and reproducibility.

3.1 Introduction

Scientific workflows are typically composed of many diverse components, each with specific dependencies against the software platform and libraries. They require multiple components to be deployed and configured before and during runtime. For a scientific method to be effectively reused over a period of time, and for experiments to reproduced, the repeatability of these deployment and configuration steps is crucial. Otherwise, the value of building workflows is quickly lost to "workflow decay" (see Chapter 1) as shown in the analysis presented in [145] that approximately 80% of the workflows used in the test cannot be either executed or produced the same results (if testable), and the earlier produced workflow had more than 80% failure proportion. Unfortunately, it is impractical to expect most scientists to perform these complex deployment steps manually as well as to deal with changes in components and dependencies.

In order to improve the reusability and portability of workflow applications and to automate their deployment we propose to base our work on an emerging OASIS standard: TOSCA, which aims to enable the automated deployment and management of cloud applications. It is designed to be sufficiently generic to cover a variety of scenarios, and enables the modeling of applications in a way that allow them to be portable between different cloud management environments [20] [67].

In this chapter we present our proposed approach for using TOSCA as a language to

describe workflows, workflow components and the execution environments. We want to offer them as reusable entities that capture not only the workflow itself but also all details needed to deploy and execute it.

In addition, we demonstrate our approach in practice by modeling an existing real workflow (we have modelled many workflows in developing and evaluating our approach, but focus on one as a running example in this chapter). The example involves a typical scientific workflow, i.e. a set of tasks with data dependencies expressed as a directed acyclic graph. We show how this approach can not only be used to represent workflow components and the workflow itself but also to capture the configuration of the whole application and the execution environment. The components of this scientific workflow will be modelled, along with the relationships that represent the connections between components. This results in a workflow template that enables automatic deployment on the Cloud, and so can dynamically improve the reusability and reproducibility of workflows.

The remainder of this chapter is structured as follows. Section 3.2 gives a detailed description of TOSCA and its components. Section 3.3 presents our approach to model the nodes and relationships in a scientific workflow. The description of a use case and details of mapping using TOSCA are presented in Section 3.4. Finally, we draw conclusions on what we have learnt in Section 3.5.

3.2 TOSCA in Detail

TOSCA is a new specification for modeling a complete application stack, and automating its deployment and management in the Cloud [20]. The main goals of the TOSCA specification are to: 1) facilitate the description of applications, including both the structure and management aspects of the application life-cycle, 2) improve the portability of cloud applications in the face of growing diversity in cloud environments and 3) support the interoperability and reusability of application components.

The specification defines a meta-model for describing both the structure and management of cloud applications. The core of TOSCA modeling is the *Service Template*, depicted in Figure 3.1, which consists of three logical parts :*Node and Relationship*

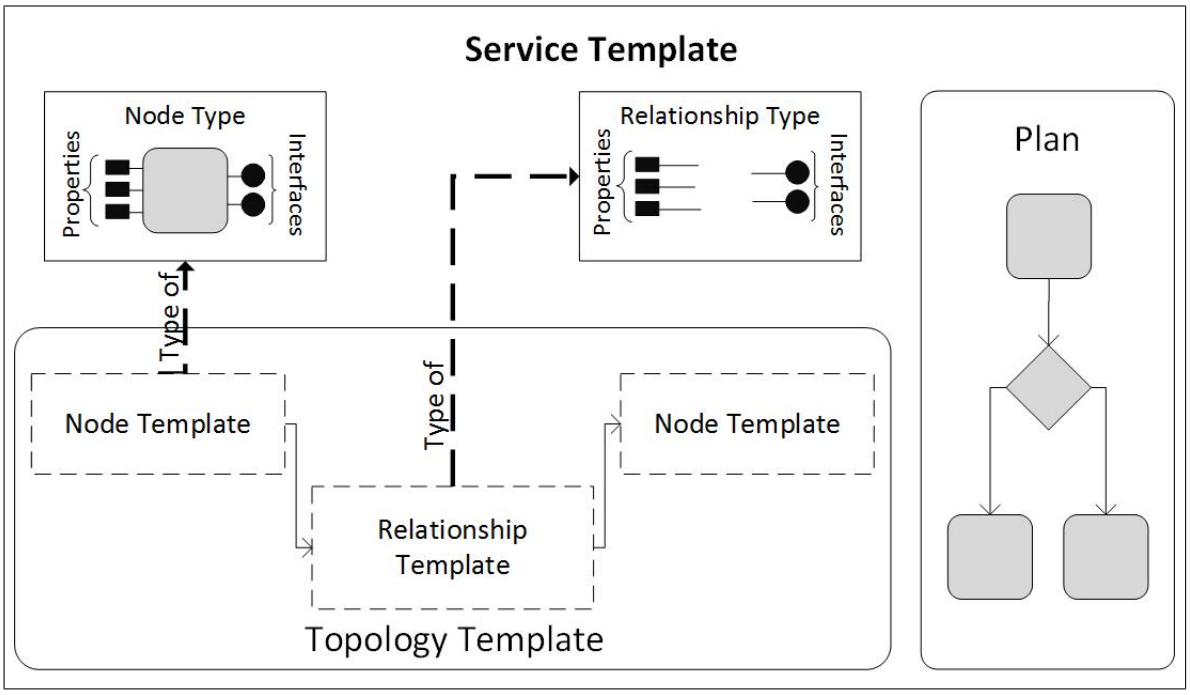

Figure 3.1: Type definitions and templates in TOSCA.

Types, *Topology Template* and *Management Plans*.

The Topology Template defines the structure of an application. It consists of *Node Templates* and *Relationship Templates*. The Node Template describes the required components and their properties, interface operations and input/output. The component's properties includes name, description, features and parameters required to configure the components. Each Node Template is an instance of *Node Type*, and defines the structure of application components including properties and operations for life-cycle management of a component. Whereas the Node Template provides exact values for the properties and implementations of the operations based on these types definitions. Node Types and Templates are defined separately to support reusability; the same Type can be instantiated multiple times to create different Node Templates in the same topology and also can be referenced by others [20].

Relationship Templates specify the relationship between nodes in the Topology Template. Similarly, a Relationship Template is an instance of *Relationship Type* which defines the properties and potential status during runtime and is used to specify a type of one or more Relationship Template. Together they are able to describe the logical relationships and other dependencies between the application's node templates [99].

The deployment process, i.e. creation, configuration, activation and termination of a service, is defined by *Plans*. The purpose of Plans is to interpret the templates and execute appropriate actions. Plans encode a sequence of operations required to instantiate TOSCA services and thus they follow an ”imperative” approach. The use of plans is not however mandatory: often, a TOSCA runtime environment is able to infer a correct deployment plan and management procedure by interpreting the service topology and tracking the relationships between the Node Templates. This is known as the “declarative” approach [20].

The main advantage of the declarative approach is that it hides low-level deployment activities from the user. Scientists can focus on the definition of the high-level structure of their experiment, which the TOSCA runtime can translate into a detailed deployment procedure. In this work we therefore adopt the declarative approach and use the Topology Template to define workflows.

TOSCA is still an emerging standard. Originally it used an XML-based modeling language for application specifications but the TOSCA technical committee has more recently approved the TOSCA Simple Profile in YAML Working version 01. This specifies a rendering of TOSCA DSL in more accessible and concise way in order to minimize the user learning curve and speed up the adoption of the standard [101].

Once the Service Template has been prepared for an application, a TOSCA-compliant runtime environment is required to deploy and manage the application. Currently, there are number of such runtime environment available including OpenTOSCA [19], Cloudify [1] and Alien4Cloud [2].

In this thesis we used vendor-specific flavour of TOSCA YAML provided by Cloudify. Cloudify is a free and open-source orchestrator platform that intends to use TOSCA to automate the deployment and scaling of applications over number of cloud technology. In addition, it provides a vendor specific flavour of TOSCA YAML - Cloudify DSL (e.g. in the DSL the Service Template is called blueprint) - a command-line interface to execute blueprints, and can use different IaaS APIs to launch applications. It supports the declarative processing of TOSCA application. Cloudify’s DSL specification allows

[1] http://getcloudify.org)
[2] http://alien4cloud.org

creating and using custom types to produce a particular blueprint that defines cloud applications capturing all components and execution environments. Currently, it is being actively developed and has a vibrant community, which makes it a promising research platform.

3.3 TOSCA-Based Modeling of Scientific Workflow

TOSCA's primary use is to model the structure of a cloud application, which may include both a horizontal 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 through a host-hosted relationship, where the host component provides an execution environment for the hosted component.

For workflows, horizontally means describing the tasks' dependencies which depict the order of tasks' execution and data transformation. Vertically provides a full software stack description of each task including the host environment (VM and/or container), required software libraries to execute the task (Wine, Java, and other special tools), the task itself and all the dependency relation among them. Usually, scientific workflow modeling merely focuses on the dependencies between tasks in the horizontal dimension, whilst the aspects related to the vertical dimension, such as creating a task's runtime environment are ignored.

In this thesis, we have addressed this gap to generate a reusable and deployable workflow description. We have devised a way in which it is possible to use TOSCA to define a workflow's structure, components together with their requirements, relationship and life-cycle management.

Our approach to workflow modeling: 1) allows the description of workflow with all required components horizontally and vertically. 2) improves workflow portability by using TOSCA to provide a new way to enable the creation of portable description of cloud applications and to automate their deployment and management. 3) supports the reproducibility of workflows by providing a reusable definition for their components as well as the whole workflow. 4) utilizes the life-cycle management operations of the node

templates such as (create, configure, start etc.) to manage a dataflow implementation of the workflow.

In this section we show how TOSCA can be used to model a scientific workflow, including discussion of the different stages followed to create a complete Service Template.

3.3.1 Modeling Workflow Building Blocks

In principle, to define a structure of any application using TOSCA one needs to model a set of Node and Relationship Types, Node and Relationship Templates, and include them in the topology part of the Service Template. XML-based TOSCA and TOSCA-based Cloudify DSL (Domain Specific Language) offers a number of base types (node types and relationship types) supported by a range of run-time environments. Therefore, Topology Templates can be either constructed using existing base types or new types can be created by customizing the existing ones. The new types are derived from the base types in support of the inheritance functionality (DerivedFrom tag) offered by TOSCA, which allows us to design application components and define several software stack layers and types (tasks, libraries and execution environments) [20].

Following this, we define a TOSCA-based description of a workflow as follow:

- defining a series of Node Types, Relationship Types for representing all workflow components and their dependencies. These definitions help to capture all entities of the workflow.
- building a Topology Template of the workflow, in which workflow tasks and task dependencies are modeled as Node and Relationship Templates, respectively.

Node and Relationship Types also include the declaration of Interfaces. An Interface defines the life-cycle management operations that can be applied to a component.

3.3.1.1 Workflow Components as Node Types

The first step to model a workflow using TOSCA is to identify all its constituent parts. These include workflow tasks and all their software dependencies such as the specific

packages and libraries required by the tasks to run. Each of these basic components and dependencies, including their interfaces and properties are described as TOSCA Node Types. Node types are usually derived from the basic node types provided by TOSCA and Cloudify DSL, like *ApplicationModule*, *Compute*, *SoftwareComponent* etc. The basic types define the set of basic components required for applications to function properly. All of these basic types are directly derived from a generic TOSCA node type called *Root*. The basic node types are then customised with specific property and interface definitions to create new node types. Furthermore, a node type can own interfaces to manage the life-cycle operations of workflow components. Central to this is the Cloudify-specific life-cycle interface with operations to create, start and terminate a service. When defining workflow components, each of them will need an implementation of the life-cycle interface.

3.3.1.2 Task Dependencies as Relationship Types

To capture dependencies between nodes, TOSCA offers a number of generic Relationship Types such as *depends_on* and *connected_to*. These types define an interface with operations to configure the source and target nodes joined by the relationship.

Among the basic relationship types one of the most common is *contained_in*. It allows a vertical software stack to be created as in the case of a virtual machine that hosts an operating system, which in turn hosts one or more workflow services. The relationship definition is used to specify the semantics of a link between nodes and also methods which realize such a link. For example, the *connected_to* relationship needs implementation of methods which can bind two end nodes, as in a client-server connection.

When connecting new, non-standard node types, a new Relationship Type should be defined that specify the connection between specific Node Types and extended one of the basic Relationship Type characterized by a source type and a target type.

3.3.2 Constructing a Workflow Topology Template

The TOSCA metamodel uses the concept of a Topology Template to describe a cloud application. We use it to model the high-level structure of scientific workflows. As

mentioned before, Topology Template is a graph of Node Templates which represent specific instances of application components and Relationship Templates that model links between these instances. Clearly, it fits the notion of scientific workflow very well.

In TOSCA-based modeling of a workflow, Node Templates in the Topology represent all workflow components while Relationship Templates defines different links between the components both vertically and horizontally.

These templates will provide the comprehensive structure of a workflow and the values for the properties and implement interface operations. Moreover, templates describe the actual instances (of components or links between components) to be created and managed in a certain workflow deployment. Again, this corresponds very well to the workflow domain where a workflow task (single type) can be included in a workflow definition multiple times (multiple templates).

Importantly, the entire Topology Template may be treated as another Node Type, which greatly improves reusability.

3.4 Use Case: TOSCA-Based mapping of a Real Scientific Workflow

To demonstrate the feasibility of using TOSCA to model scientific workflow applications we selected an existing workflow (Neighbour Joining *NJ*) that performs phylogenetic analysis of the Leishmania parasite. The workflow is used in the EUBrazil Cloud Connect project [3] to perform identification of Leishmania species using the neighbour joining method. Originally, it was designed in the e-Science Central system (e-SC), and we now present its specification as a TOSCA service template. The result is a self-contained and portable service model that can be used to deploy and manage workflow instances in the cloud.

Figure 3.2 depicts the selected workflow as designed in e-SC. It consists of 11 blocks of which 9 are Java-based and 2 others (ClustralW and MEGA-NJ) wrap executable tools to perform sequence alignment and the neighbour-joining analysis. In addition, the

[3] http://www.eubrazilcloudconnect.eu

blocks required different software dependencies such as Java Runtime Environment (JRE), some specific libraries for experiment processing. The MEGA-NJ task is a Windows executable - to be executed in Linux it requires the Wine library.

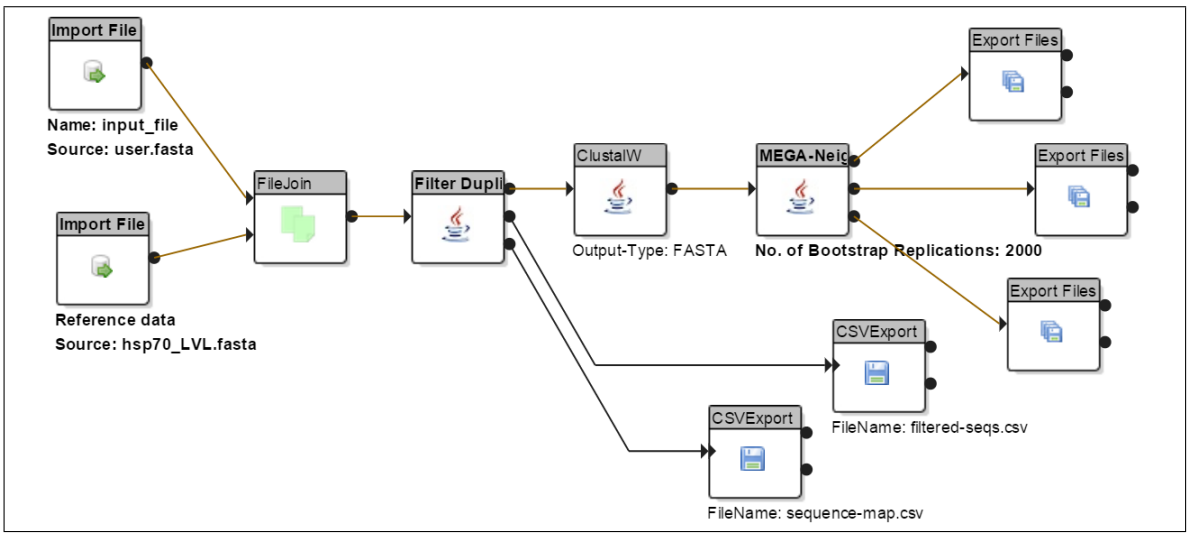

Figure 3.2: An e-SC modeling for Neighbour Joining *NJ* workflow.

Following the concepts of the TOSCA specification, the topology description and components definitions for the *NJ* workflow can be modeled as follows.

3.4.1 Workflow components as Node Types

There are two types of components in e-SC workflows – blocks and shared libraries needed by the blocks. To describe them we defined two node types. From these two types we derive node types corresponding to all e-SC workflow blocks and libraries in the example; this two level hierarchy is depicted in Figure 3.3:

- Specific Node Types: Nodes at this level represent the most fundamental part of any e-SC workflow. They are derived from the Cloudify basic node types and define: (1) a generic workflow block that offers common properties to all types of blocks, and (2) a generic library that forms the basic type for all shared libraries in a workflow. We derived them from the *ApplicationModule* node type defined in Cloudify, which is a base type for any software module or artifact to be deployed. Listing 8.1 presents the complete node type definition for a workflow block.

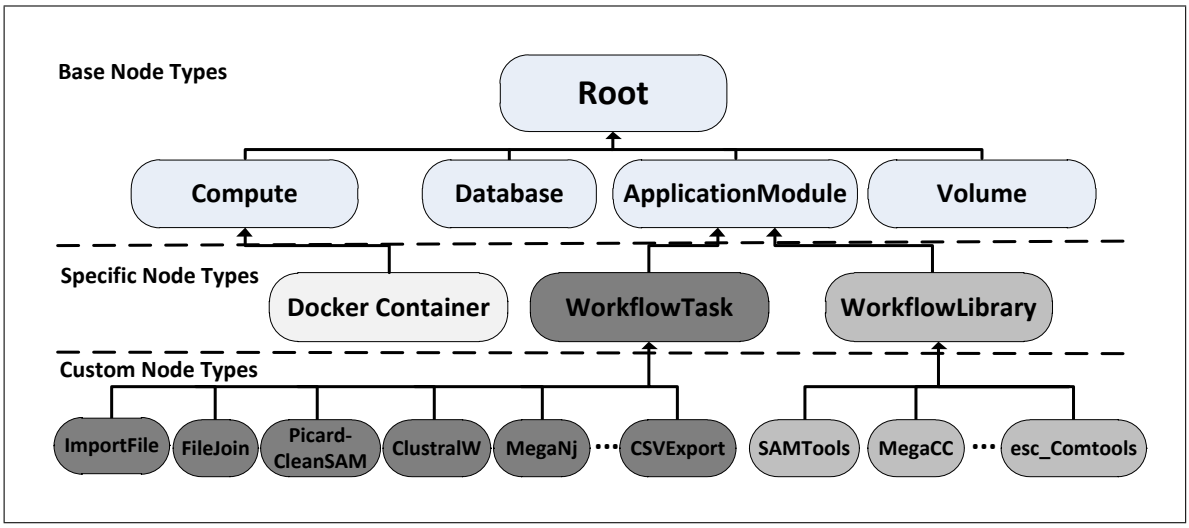

Figure 3.3: Node types hierarchy for modeling scientific workflow.

Listing 3.1: Node Type definition of a workflow block.

```
workflow_task:
  derived_from: cloudify.nodes.ApplicationModule
  properties:
    block_description:
      description: Description of task function
      type: string
    block_name:
      type: string
    block_category:
      type: string
    service_type:
      type: string
```

- Custom Node Types: This node type is used to represent particular types of workflow blocks and includes information about their specific properties, configuration, and block inputs and outputs. In addition, node types for any required library are defined such as run-time environments required to run a specific service (e.g: java run-time, R run-time, and etc.) and special libraries (e.g: Core e-Sc and MegaCC). Custom node types will be instantiated by node templates to represent the actual blocks and libraries that compose a specific workflow. Listing 3.2 presents an example of the node type for a selected e-SC block.

  In TOSCA, Node Type definitions are reusable and may be referenced in other workflows and by other developers. Thus, to facilitate reuse we store them in our Node Type Repository to make them available for future use.

Listing 3.2: Node Type of custom workflow block *FilterDuplicate*.

```
FilterDuplicates:
    derived_from: workflow_service
    properties:
      arg-Normalize_Sequence_Names:
        type: boolean
        default: false
      arg-Normalize_Duplicates_Only:
        type: boolean
        default: false
      # input ports
      input-fasta-files:
        type: string
        default: file-list
      # outputs ports
      output-filtered-fasta-files:
        type: string
        default: file-list
      output-removed-sequences:
        type: string
        default: csv-data
      output-sequence-map:
        type: string
        default: csv-data
```

3.4.2 Block dependencies as Relationship Types

Most of the relationships used in an e-SC workflow are common to any cloud application. For example, the *contained_in* relationship may denote that a block is hosted on a VM or container. The exceptions, however, are data dependency links that connect block input and output ports. We derived them from the generic *depends_on* relationship type to implement the *preconfigure* operation that will pass data between connected blocks.

The exact semantics of the task link depends on a specific workflow management system and for this reason we defined the new relationship type *task_link*. It is derived from the generic *depends_on* type and specifies the implementation of the life-cycle operation related to the task link. The operation is responsible for the actual data transfer between connected tasks. For the *task_link* relationship, we defined the preconfigure operation that will perform the actual data passing between connected tasks. This type will be used to create relationship templates between specific connected tasks in a workflow, while the life-cycle operation will be used to transfer data between tasks.

Listing 3.3 shows one of the new relationship types.

Listing 3.3: The definition of the *task_link* Relationship Type.

```
task_link:
    derived_from: cloudify.relationships.depends_on
    source_interfaces:
      cloudify.interfaces.relationship_lifecycle:
          preconfigure:
              implementation: scripts/preconfigure.sh
```

3.4.3 Constructing the NJ *Workflow Topology Template*

The TOSCA metamodel uses Topology Template to describe a cloud application. While it is intended to describe service-based systems in which services execute for undetermined periods, we can impose the sequential execution of tasks by adding the *task_link* relationship between components. This forces the runtime environment to deploy and execute components in the order implied by the dependencies.

As mentioned earlier, the Topology Template is a graph of Node and Relationship Templates. The former represents specific instances of application components, whereas the latter models links between these instances. Clearly, it fits the notion of scientific workflow very well – Node and Relationship Templates are instances of Node and Relationship Types much like workflow tasks included in the workflow are instances of one or more tasks available in the task repository. Moreover, if types declare properties and interfaces, templates provide values for the properties and implement interface operations. Again, this corresponds very well to workflow modeling where a workflow task from the repository (Node Type) to be included in a workflow has to be properly configured (become a Node Template). Further, the entire Topology Template may be treated as another Node Type, which improves reusability and flexibility in workflow composition.

Following the TOSCA topology template rules, we present in Fig. 3.4 the topology of the Service Template which represents the *NJ* workflow given earlier. This includes not only the high-level structure of the workflow (i.e. task dependencies) but also all library dependencies and the deployment of components in containers and virtual machines. Thus, we can capture the complete software stack required to deploy and enact the workflow:

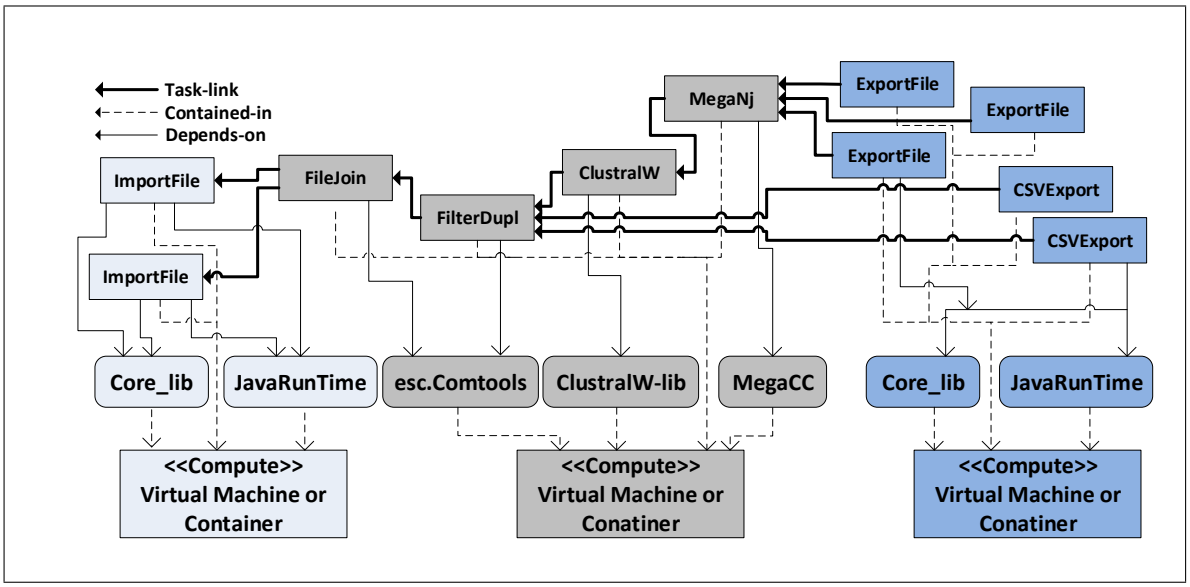

Figure 3.4: TOSCA Topology Template of the *NJ* Workflow.

- The task execution environment is described by means of the VM and container Node Templates. This allows the definition of the properties of the VM and Docker image required to run a task or group of tasks.

- The dependency libraries are described as Node Templates and include details about the type, version and URL of the library artifacts. All libraries are linked to the underlying execution environments by instantiating a Relationship Template from *contained_in* type.

- Workflow tasks and their data dependencies are represented as Node and Relationship Templates. They include specific property values together with input and output ports and life-cycle operations. Each task instantiates a number of Relationship Templates implementing its links with other tasks, dependency libraries and the host environment using *task_link*, using *depends_on* and using *contained_in* Relationship Types respectively. Listing 3.4 shows the Node Template for one of the workflow tasks. For the complete Topology Template of a workflow consisting of three tasks, please see Appendix A.

Listing 3.4: Node Template definition of a selected workflow task.

```
FilterDupl:
    type: filterDupl
    # tasks properties required to execute the task
    properties:
      block_description: "Filter duplicated sequence reads in
         the FASTA format."
```

```
  block_name: filterduplicates.jar
  block_category: File Management
  service_type: block
# Relationship Templates instantiated from the defined
   Relationship Types
relationships:
    # Relationship to connect the task with the host
       container
  - type: cloudify.relationships.contained_in
    target: container4
    # Relationship to connect the task with the a required
       library
  - type: cloudify.relationships.depends_on
    target: Java3
    # Relationship to connect the task with a predecessor
       task "FileJoin"
  - type: block_link
    target: FileJoin
    # The relationship lifecycle operation to obtain data
       from the previous task
    source_interfaces:
      cloudify.interfaces.relationship_lifecycle:
        preconfigure:
          implementation: Core-LifecycleScripts/datacopy.sh
          inputs:
            process:
              args: [FileJoin/file-3, FilterDupl/fasta-files
                 , NJ, container4]
# The lifecycle operations for task management
interfaces:
  cloudify.interfaces.lifecycle:
    # downloading the task to the hosted container using the
        provided URL and creating a corresponding task image
    create:
      implementation: Core-LifecycleScripts/task-download-
         multi.sh
      inputs:
        process:
         args: [{ get_input: create_image }, container4, '
            https://github.com/rawaqasha/eScBlocks-host/raw/
            master/filterduplicates1.jar']
    # delete the previous task after obtaining the required
       data
    configure:
      implementation: Core-LifecycleScripts/containers-clean
         .sh
      inputs:
        process:
         args: [container3]
    # start the execution of the task on the host container
    start:
      implementation: Core-LifecycleScripts/task-deploy.sh
      inputs:
        process:
          args: [NJ, container4]
```

3.5 Conclusion

The ability to package cloud applications in a way that enables their reusability and portability is an important precondition to truly realizing the benefits of Cloud computing for scientific and other applications. It does, however, require the existence of a well-defined specification that allows us to capture complex deployment and configuration requirements.

We have therefore presented a new approach that uses TOSCA to describe formally the internal topology of a scientific workflow in a comprehensive and portable way, together with its deployment processes.

By defining reusable Node Types for workflow tasks and Topology Templates (blueprints) for complete workflows, we enable the description of workflow. Additionally, TOSCA supports composition, and so we can nest complete workflows one in another, which not only gives flexibility at design time but also improves reuse. Overall, building scientific workflows using this approach has a beneficial impact on the reusability and portability of workflow applications as we will demonstrate in the next chapters. In particular, we will show how this can enable automatic deployment and scalability.

As a potential benefit, employing TOSCA in description of workflow enables valuable use cases such as portable services and workflow to achieve automatic deployment and scalability.

In the next chapter, our approach is integrated with Docker technology to build a new framework for the dynamic deployment of scientific workflows.

4

Dynamic Deployment of Scientific Workflows in the Cloud using Container Virtualization

Summary

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 chapter we describe a solution - a system that allows workflows to be portable across a range of clouds. This portability is achieved through a new approach for building, dynamically deploying and enacting workflows. It combines the TOSCA-based specification method described in the previous chapter with container-based virtualization. We describe a working implementation of our approach and evaluate it using a set of existing scientific workflows - this illustrates the flexibility of the proposed approach.

4.1 Introduction

The scalability and ability to acquire resources on-demand offered by Cloud computing makes it attractive for workflow management [149]. In addition, cloud offers the opportunity to share, exchange and reuse services and experimental methods [53]. However, efficiently meeting workflow requirements in the cloud requires addressing key issues in the provisioning of the execution environments, and subsequent workflow execution [136]. 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 [145].

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 of 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, this chapter presents a new approach to describe, build, dynamically deploy and enact workflows on the Cloud. We extend our modeling approach proposed in Chapter 3 and presented in [107] and implemented the provision and deployment of workflows in a way that significantly increases their portability.

The approach integrates the TOSCA standard with container-based virtualization. TOSCA supports the description of cloud applications in a portable way [20], which we exploit to allow heterogeneous workflows to be deployed in the Cloud. Container-based virtualization offers the opportunity for rapid and efficient building and deployment of lightweight workflow components [45], and we also use it to isolate task execution. In this work, we use Docker containers to dynamically provision the execution environment and construct the full software stack required by a workflow component or group of components. This allows 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 use TOSCA 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. Our new approach is generic enough to cover a variety of scenarios which we use for evaluation in the following sections.

In this chapter we present a significant development of our TOSCA-based modeling

approach, we introduce the following capabilities:

- using the proposed modeling approach to describe and implement the provisioning and deployment of workflows across different cloud infrastructures, thereby ensuring application portability.
- using our approach to construct and dynamically deploy the full software stack required by a workflow component or group of components.
- exploiting container-based virtualization to improve deployment portability and isolate the execution of heterogeneous workflow components.
- automating the deployment and dynamically provisioning a selection of scientific workflows using containers.
- supporting a range of deployment options for efficiency and security isolation.

The rest of this chapter is structured as follows: Section 4.2 presents the requirements necessary for workflow deployment. Next, the details of our new approach are presented in section 4.3. The features supported by the integration of TOSCA and Docker are discussed in section 4.4. In section 4.5, the evaluation of our solution is presented. Finally, Section 4.6 closes the chapter with conclusions.

4.2 Workflow Deployment Requirements

A number of requirements need to be considered in the development of an automatic deployment system for scientific workflow.

Describing Workflow/tasks: The first requirement is offering a description of the workflow tasks and the detailed specifications of its requirements, dependencies, input/output parameters and properties; including the ability to describe components developed in various technologies.

Since a workflow is an aggregation of a set of diverse, connected tasks, its description will be the combination of tasks requirements and how they are connected together

to form a functioning system. Also the dependencies between workflow tasks must be described to determine the order in which tasks are started and executed.

Preserving the description along with the workflow and its associated tasks can improve the ability to rerun the workflow and reproduce the same results. Achieving this was the subject of the previous Chapter.

Describing the Environment and Infrastructure: To allow the adaptation of deployment to the target platform, descriptions are required for the target topology, capabilities, and available resources. Since each workflow task has its own specific requirements, it is essential to provide a full description of the deployment environment by including execution domain resources with their relevant properties and capabilities - workflow and task requirements must be satisfied by the capabilities of the target domain.

Execution Isolation: The heterogeneity of workflow services implies that they require different dependencies and various collections of software stack which may cause conflicts. For example, a task might require to be executed in a Linux system while the other needs Windows, or two tasks might demand different versions of R libraries. In such cases there must be an ability to support isolation for service execution to insure minimal disruption of the other services in the service based system. The level of isolation may depend on various criteria depending on the isolation level required between workflow services. For example, isolation is required to protect the execution of two workflows from each other. It may also be needed if two workflow services need different versions of the same shared library.

Deployment dependencies: An efficient means is required to deploy and configure all tasks dependencies, such as software packages and libraries required to execute the tasks. Without access to the required libraries, task execution will fail. One possible approach to preserve software dependencies is to record sufficient information about each software and library and then provide an automatic means for on-demand installation and configuration during workflow deployment.

Wiring of workflow components: In order to deploy a workflow two types of wiring - the links between workflow components - are required: 1) vertical wiring which

is used to link each task to its dependencies such as the software required to run the task and the execution environment, and 2) horizontal wiring, which refers to the data links required between tasks to enable the flow of data during workflow task execution.

4.3 Dynamic Deployment of Scientific Workflow

This section presents our system for scientific workflow deployment, focusing on the features that make it simple and efficient for workflow management developers. Furthermore, it describes how TOSCA-based modeling can be integrated with Docker technology to achieve dynamic and automatic deployment for scientific workflows and details about the necessary scripts required to manage the life-cycle of workflow components. Finally, the details about the full deployment process will be described.

Our approach for the deployment and enactment of workflows is depicted in Fig. 4.1. It has been implemented as a set of the reusable components and packages that reside in software repositories, so they can be downloaded to the workflow execution node, and also shared between users.

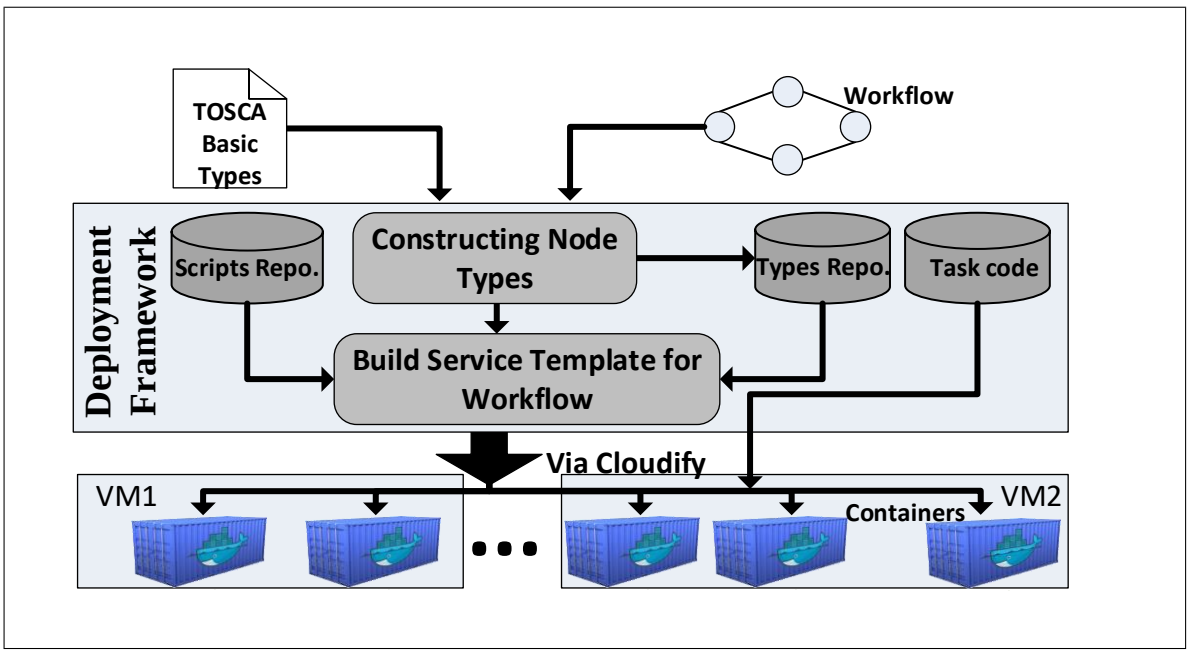

Figure 4.1: Steps from the definition to the enactment of a workflow.

Firstly, to build a workflow, we follow the TOSCA-based approach described in the previous chapter and presented in our published paper [107], and prepare basic workflow components: Node and Relationship Types, and then a Topology Template which

includes Node and Relationship Templates. Types are used to describe workflow components (tasks and their dependencies), whereas the Topology Template describes the overall structure of the workflow. It contains Node and Relationship Templates to denote all instances of workflow tasks, library dependencies and the execution environment together with the container and VM.

Further, in the template we also include life-cycle management scripts and references to software artifacts. The scripts implement the deployment actions of workflow tasks and are available in our life-cycle 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 and their dependencies. The artifacts are stored in our Task Repository to be reused across different tasks and workflows.

Finally, to deploy and enact a workflow we submit the Service Template together with the scripts and artifacts to a TOSCA runtime environment. Although 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-step 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 main steps and discuss the data exchange mechanism implemented by the approach.

4.3.1 Building the Workflow Topology

Usually, the modeling of scientific workflows focuses merely on the horizontal dimension – the data dependencies between tasks – while important aspects related to the vertical dimension, such as creating tasks' runtime environment are ignored. These are however crucial for improving workflow portability and reproducibility. Therefore, we use TOSCA, as described in Chapter 3, to model the structure of a workflow in both dimensions - the horizontal space and the 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.

Although TOSCA was intended to describe service-based systems, we impose data dependencies and the sequential enactment of tasks using the dependency relationship between components. These relationships inform the runtime environment of the appropriate order of execution. The runtime cannot initiate the deployment of a task unless all tasks that it *depends_on* have already been completed.

4.3.2 Managing the Workflow Deployment Life-cycle

Both node and relationship types define life-cycle operations to implement their activities. These are invoked by the run-time environment when deploying the workflow. For example, a node type for a task might provide a 'create' operation to handle the creation of an instance of the task at runtime, or a 'start' or 'stop' operation to handle a start or stop activity. These life-cycle operations are supported by implementation artifacts such as scripts attached to nodes and relationships. The run-time environment processing a TOSCA service template uses these life-cycle operations to instantiate components at runtime and also to pre- and postconfigure relationships and derive the order of component instantiation.

We implemented them as a set of generic scripts that can:

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

- destroy a task container and remove any redundant intermediate data.

As these scripts are reusable across a range of workflows and tasks, we store them in our Life-cycle Script Repository,[1] so they can be easily included in any newly designed workflows. We refer repository URL in all the example workflows that we use to evaluate our approach.

4.3.3 Task Deployment using Container Virtualization

Once the workflow Service Template, life-cycle 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.

Our deployment approach supports two scenarios for workflow deployment: a single container for deploying the whole workflow and multiple containers for isolated deployment of workflow tasks.

In the single container scenario, all tasks, and their dependencies are provisioned in a single container and a single image is used to create the container hosting the whole workflow. Figure 4.2 depicts the details of deploying a workflow. The process starts by creating a Docker container as a workflow execution environment, which requires pulling the specified image from Docker Hub. The user may use a generic image - available from the Docker Hub[2] or pre-built image generated by the user which includes libraries and software required by all workflow tasks. The next step is the installation and configuration of the dependencies required by all tasks which demands package downloading or on-line installation. Then, each task artifact is downloaded from a repository (in our case, Git Hub). This is followed by retrieving the data to be processed by the task, then task execution. The operations of task downloading, data retrieving and task execution are repeated for all tasks in the order specified in the Topology Template and finally, the hosted container is destroyed.

[1] https://github.com/WorkflowCenter-Repositories/Core-LifecycleScripts
[2] https://hub.docker.com

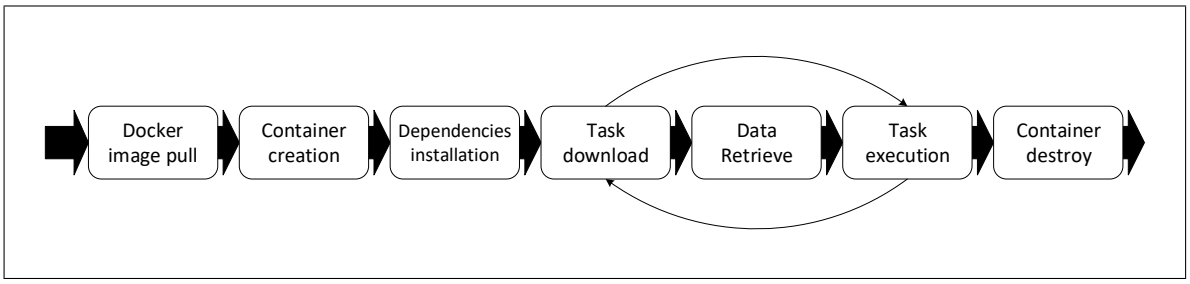

Figure 4.2: The single container workflow deployment

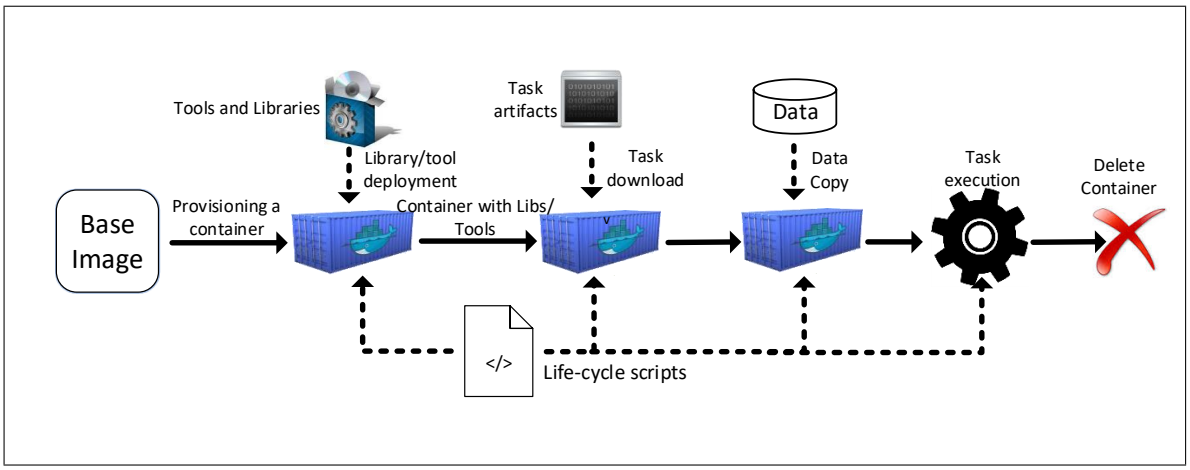

Figure 4.3: Isolated deployment of a workflow task.

In the second scenario, the isolated deployment of the tasks requires the provisioning of each task in a separate container. The tasks, dependencies and the hosted containers are provisioned in an order specified in the Topology Template of the workflow. In this scenario, each workflow task follows the deployment process shown in Fig. 4.3.

First, a Docker container is created using a task image indicated in the Topology Template. As in the first scenario, the image may be generic or it may be generated by the user and includes libraries and software required by the task.

Once the task container is running, the installation of dependency libraries takes place according to their order in the Topology 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, any input data required to run the task is copied to the container and that is followed by downloading task artifacts into the container. Similar to the previous step, this downloading step may be avoided if the selected Docker image already includes the required files.

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 Repository. This allows us to freely update tasks and to minimize the number of specific Docker images held in the repository.

Finally, with all prerequisites in place, task execution is initiated, and upon its completion the output data is transferred out of the container. Currently, to pass data between containers we use a mounted shared folder that is accessible by all containers running in the host VM. Thus, when data is generated by a task in one container, it is immediately available to other containers and so can be used as input for subsequent tasks. If the completed task is the last task to be executed in that container, the container is also terminated and deleted.

Our deployment approach supports the isolation execution for each task, however it does not offer parallel provisioning for unrelated tasks that have no relationship to each other. The reason behind this limitation is that our deployment approach uses Cloudify as a run-time environment for TOSCA-based workflow modelling, and it does not offer parallel execution. While, as shown in figure 3.4, our TOSCA-based approach could be used to model number of unrelated tasks, e.g. the two ImportFile, the three ExportFile or the two CSVExport, in an independent way so they can be run in parallel if the run-time environment supports this capability. For example, the two ImportFile tasks have no relationship to connect them together and they have been modeled as independent tasks but Cloudify unable to manage them in parallel instead it sequentially executes the tasks lifecycle operations, starts from the first task, completes all of its management operations and then starts the second instance of ImportFile task.

In addition, our approach can support another optional scenario which is hybrid of the above two scenarios. In this scenario, multiple containers can each host multiple 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 as specified in the Service Template according to tasks' dependencies and isolation requirements. Then, the approach will maintain the appropriate order of execution while reusing the first container to run task T_3. This gives the workflow designer freedom in planning how tasks are distributed across

containers without affecting runtime effectiveness.

In a case of any kind of failure occurs during the deployment time, e.g.: a failure to create a container, network disconnection during the download of dependency library or task execution failure, scientists/users will have number of artifacts and some information that can be used to diagnose the cause of the failure and help redeploying the workflow. Firstly, during the deployment process, our system keeps all the intermediate data produced by each task under a folder with task name and created during task provisioning. Therefore, when the provisioning of a task fail, it is possible to identify the broken task from the last folder of task data been created. Secondly, after the completion of task execution and transforming its output to the next task, the task container is automatically destroyed and only the container of the current running task exits. When a failure occurs, only the container of the failed task will exist in the system which indicates the failure step and should be manually deleted in order to start a new workflow deployment. Thirdly, Couldify provides an adequate log for the deployment process capturing the execution information of each lifecycle operation for both Node and Relationship Templates which can be used to diagnose and identify when and where the error occurred.

4.3.4 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 work, if all containers are deployed in the same VM, we use that host VM's disk to store input/output data. We also use this disk as a shared space to exchange data files between tasks. In this way we can minimise overheads related to transferring input/output files and data between tasks deployed on the same machine and avoid the overheads of downloading input data and uploading outputs to/from an external repository.

If tasks are deployed over multiple VMs then we use a direct network connection to transfer data between tasks. 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 life-cycle script that implements inter-task dependency relationship, and it remains hidden from the workflow designer.

4.4 The Integration of TOSCA and Docker for Workflow Deployment

The integration of the TOSCA-based modeling approach presented in the previous chapter and Docker technology for addressing the automatic deployment of scientific workflow brings number of valuable deployment features as presented below.

4.4.1 On-demand deployment and Pre-built Docker Images

Our system provides the flexibility to provision a particular task in a number of different deployment scenarios, yet all using the same high level workflow definition. The task description is sufficient to support the following two scenarios: 1) the on-demand (dynamic) scenario using an image containing merely a basic OS. In this case, the TOSCA-based task description provides all the information required to build the software stack needed to execute the task following the steps shown in figure 4.3. 2) pre-built image scenario, where a Docker image packages all dependencies needed for task execution and is ready to be used for running the task immediately. In this scenario, the task description is still useful: it provides the means to check the availability of task dependencies and artifacts.

Moreover, our TOSCA-based description is flexible enough to adopt different scenarios for provisioning tasks within the same workflow. For example, some of the tasks can be dynamically provisioned on-demand while the others are provisioned using pre-built images as demonstrated in experiment in section 4.5.5. This flexibility in the deployment of tasks is efficient and appropriate to address the challenge of irregular changes in tasks and their dependency libraries. Therefore, tasks that are frequently changed can be provisioned using the on-demand scenario while for unchanged tasks an image can be created once and used subsequently to avoid re-installation of the required libraries and unnecessary downloading of task's artifacts.

A Docker image can also be used to package any number of tasks and their dependencies if there is no conflict among these requirements. As a consequence, provisioning scenarios using the pre-built images support 1) the flexibility of task provisioning where different images can be used to provision the same task with different levels of settings, 2) reducing the total deployment time when the same image can be used to package N tasks sharing the same requirements, where only one image is downloaded, rather than N images for N tasks and 3) performance efficiency in the automatic adoption of different deployment scenarios for a task using the same description.

Similar scenarios (on-demand and pre-built image) can be adopted for the deployment of the whole workflow using the above-mentioned workflow topology. In the case of on-demand deployment, a base image is used and TOSCA description provides the details for provisioning tasks with their dependencies during the deployment time. While in the other scenario, a pre-built image packaging all workflow components and dependencies is used so that workflow tasks are already ready for execution.

4.4.2 Single- and Multi-Container Deployment Scenarios

In the rest of this thesis, we explore and compare the two main scenarios described in 4.3.3: a single container for deploying the whole workflow and one container per workflow task. The first scenario offers the shortest time for workflow deployment when the tasks share dependency libraries and software, and when there is no conflict between requirements. In addition, all tasks have to share the same execution environment, i.e. the operating system that hosted the tasks. The tasks provisioned in this scenario are not isolated, and there might therefore be security problems. However, this deployment scenario usually demands fewer resources such as disk storage and memory because only one image is required to create a single container to host the deployment of the workflow. In addition, the common libraries and software required by number of tasks are only installed once and shared by all the tasks.

On the other hand, the multi containers scenario provides an isolated secure environment to provision each task separately, which enables the use of different execution environments for the tasks (e.g.: different Linux distributions, Windows, etc.). In addition, it enables the deployment of different and even conflicting software depen-

dencies. We now conduct experiments to understand how big are the extra computing resources this scenario consumes due to the use of multiple tasks.

4.5 Experiments and Evaluation

To validate our deployment system, we conducted a set of experiments in which we deployed selected workflows, originally created in eScienceCentral. 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 the time required to deploy workflows in local and public cloud environments. 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 pre-build 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 repositories.[3] Although in this work we focus our discussion on a single VM host with multiple containers, our approach can be also used to deploy workflow tasks on different VMs.

4.5.1 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 4.1 summarizes the basic properties of the workflows.

Table 4.1: 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
Column Invert (CI)	7	Java, Core-lib
File Zip (FZ)	3	Java, Core-lib

The *Neighbor Joining* workflow (NJ), as presented in the previous chapter, is a pipeline

[3]https://github.com/WorkflowCenter-Repositories

used in the EUBrazil Cloud Connect project to perform species identification of Leishmania parasite and sandflies using the neighbor-joining method.

The *Sequence Cleaning* workflow (SC) (see figure 5.6) is one of the steps in the Next Generation Sequencing pipeline implemented in the Cloud-e-Genome project [29]. It consists of eight tasks of which seven are Java-based and one is a wrapper around the SAMTools executable. Finally, *Column Invert* (CI) and *File Zip* (FZ) are simple workflows that consist of only Java tasks to invert a matrix and to compress and decompress files.

To provision the execution environment for workflow tasks we used different Docker images to run the containers (Table 4.2). 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.

Table 4.2: 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

Although our approach allows us to provision containers on different VMs, all containers used throughout the experiments were deployed in a single virtual machine. And prior to workflow deployment all Docker images required by tasks were cached in this host VM, so that results were not impacted by image transfer times from the Docker Hub. 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.

4.5.2 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 5.1.

Table 4.3: Execution environments.

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

In the experiment, each of the four test workflows was deployed ten times and we used exactly the same four blueprints in each platform. Figure 4.4 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.

This experiment shows that our proposed approach is able to successfully support workflow deployment on several cloud platforms. We used the same Topology 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.

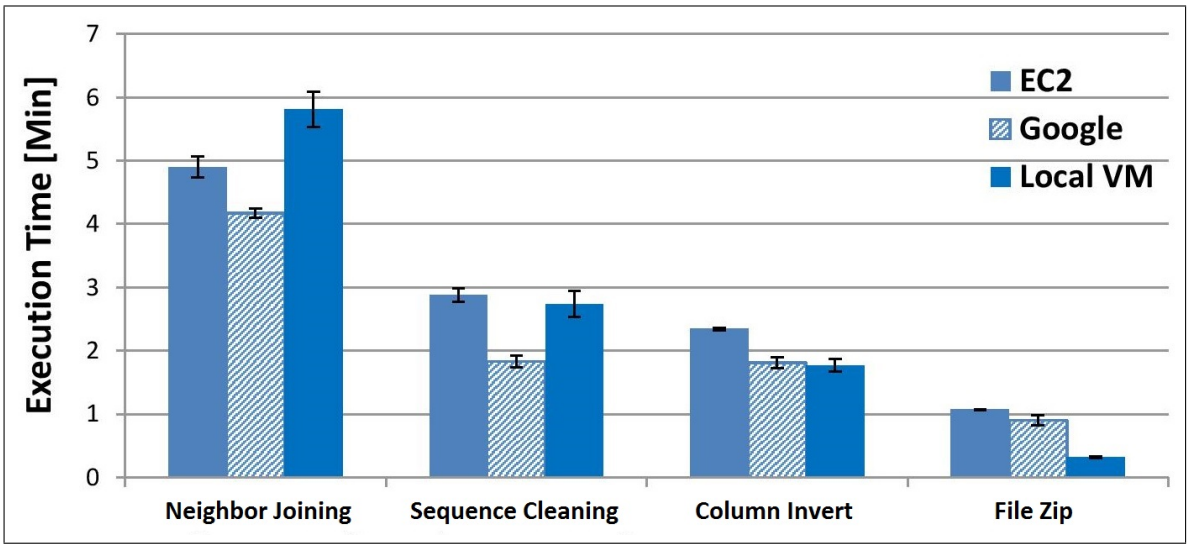

Figure 4.4: Execution time for workflows enacted in different environments; the NJ workflow used the Basic and CentOS images, other three workflows used the Basic image only.

4.5.3 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 scenarios. 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 that included tools and libraries common to most of the tasks (java and wget). Therefore, ET included the provisioning of Docker container(s), installation of task specific dependency libraries and the actual execution time of all workflow tasks.

Fig. 4.5 presents the average execution time for the four workflows. As shown, there is little difference between the two deployment scenarios for the four workflows: 14.9, 33.4, 13.9 and 3.2 seconds; or only about 2 seconds overhead per task. It reveals that

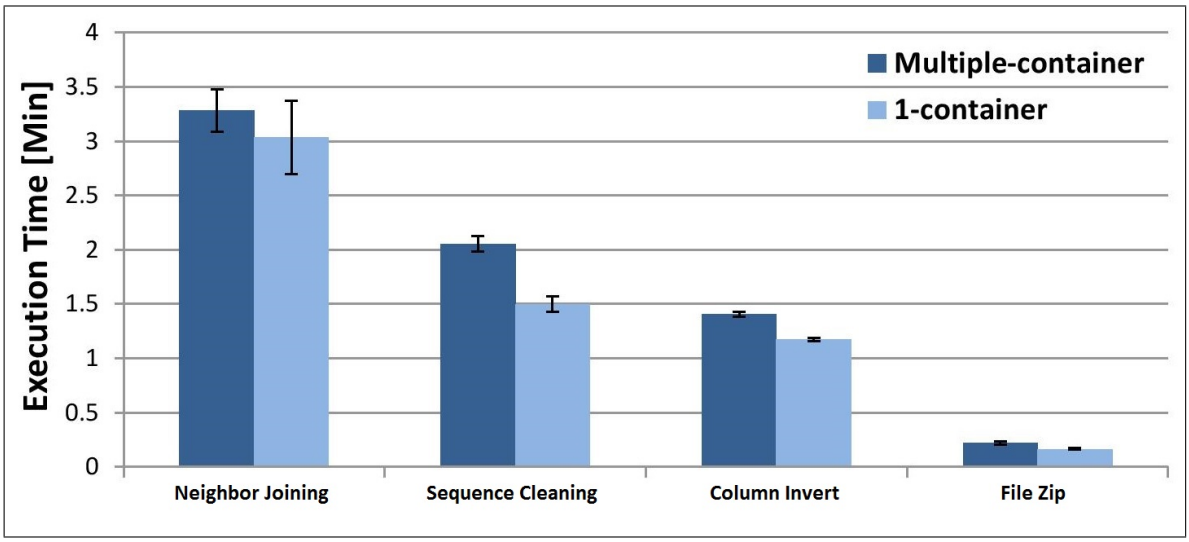

Figure 4.5: Average execution time of single- and multi-container workflow deployments; all workflows used the Basic image.

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 due to variation in the network throughput. This time the execution of the SC, CI and FZ workflows was faster than in previous experiment and the difference stems from the faster download time for task and library artifacts because they were conducted on different day time.

Running each task in a separate container not only improves the execution isolation properties but it can also improve resource usage. If a single task in a workflow requires many libraries and artifacts to be installed, with the single-container approach they would consume disk space and memory of the container until the end of the workflow.

Instead, when using the multi-container strategy, each container includes only a subset of libraries and artifacts required by the task it is hosting. And, once the task has completed, the container may be destroyed which frees the resources allocated.

4.5.4 *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 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. Having the images prepared, we used them to run the workflows in the multiple-container mode and compared the execution time with multiple-container executions using the *Ubuntu:14.04* image. By using these specialized images we avoided the download and installation of any libraries and task dependencies during workflow execution. Fig. 4.6 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.

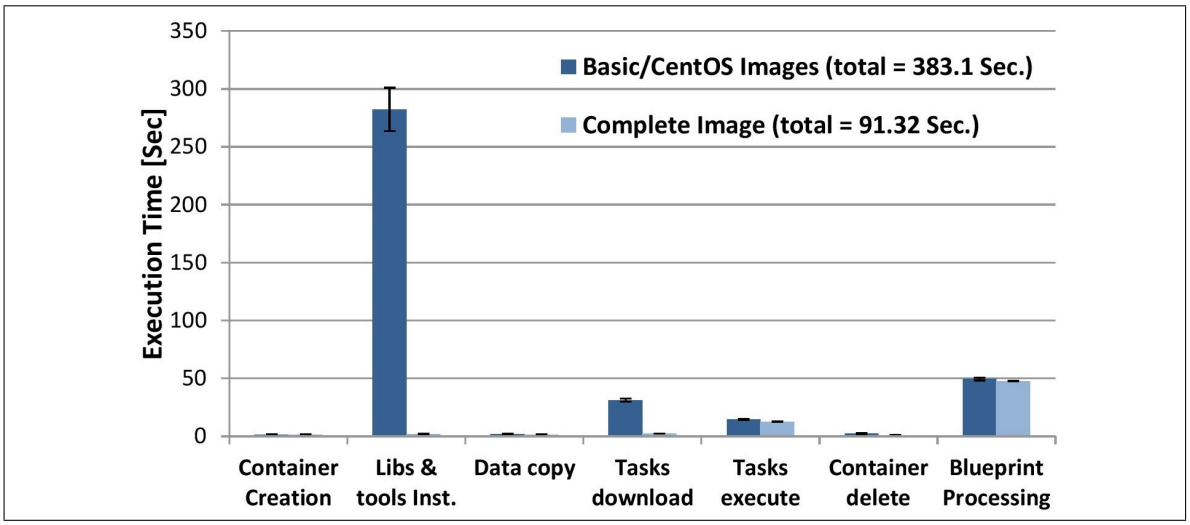

Figure 4.6: Execution time for the steps in deployment the NJ workflow.

In Fig. 4.7 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 life-cycle management scripts).

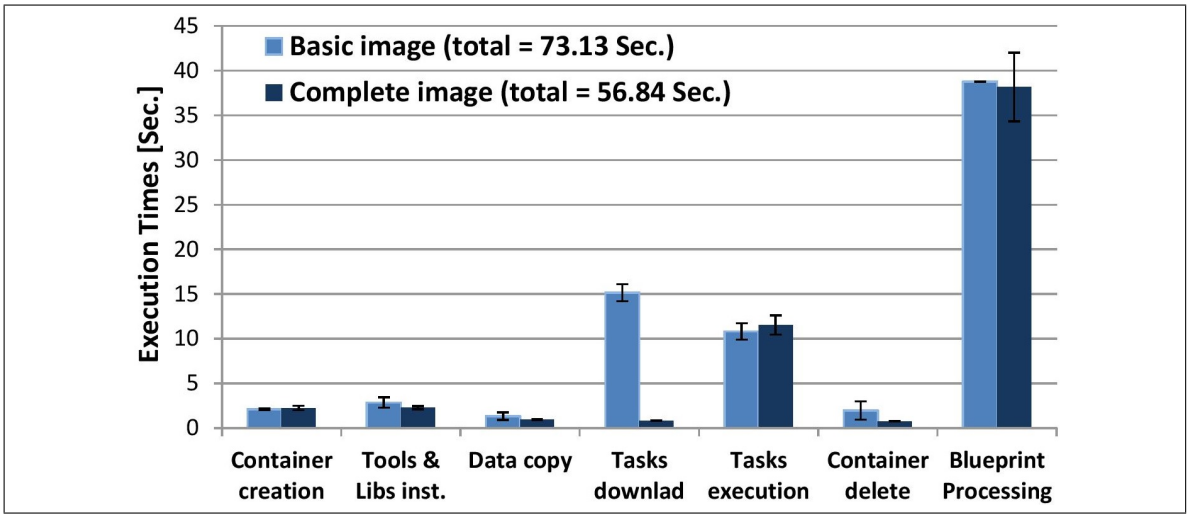

Figure 4.7: Execution time for steps in deployment of the SC workflow.

For the *SC* workflow (8 tasks) the 'blueprint processing' time was about 38 seconds, and about 10 seconds shorter than for the other *NJ* 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 and is a valuable scenario for deploying tasks subject to frequent changes.

4.5.5 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 approach is flexible enough to adopt other options to task and workflow deployment. The flexibility can help to address common challenges faced by the designers during the workflow development phase – frequent and irregular changes

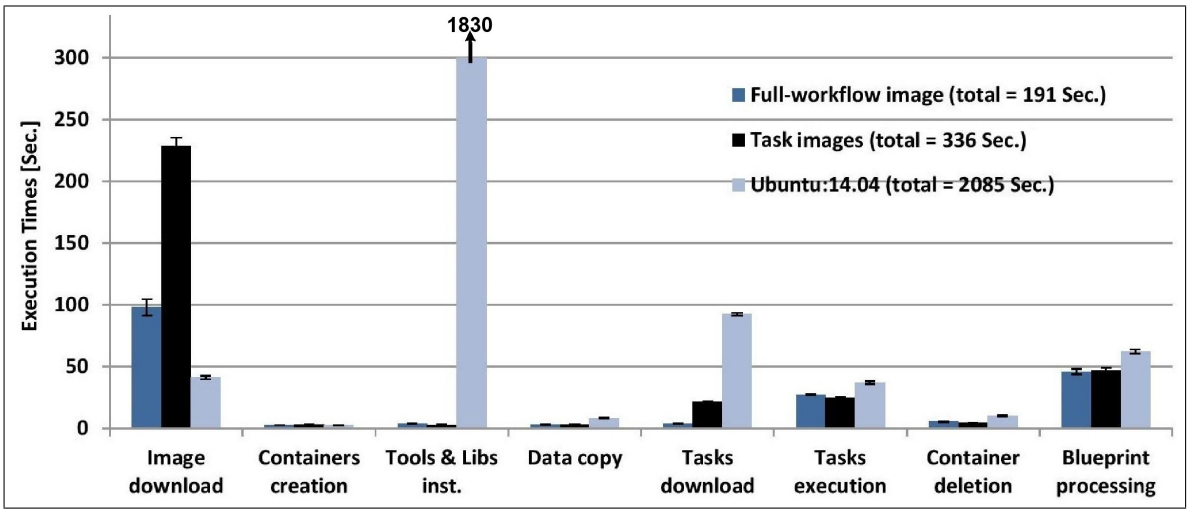

Figure 4.8: Execution time of the *NJ* workflow using three possible workflow deployment options.

in the implementations of tasks, for example, when a new interface parameter is added to the task or performance optimization made.

Therefore, in this experiment we investigate how various deployment options that are enabled by our approach can support workflow development. We look at them from two perspectives: the workflow level 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: the first used the pure-OS Ubuntu:14.04 image from DockerHub, the second 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), while the third used the *CompleteNJ* image with all dependencies and artifacts preinstalled. Fig. 4.8 shows the execution time in all three cases.

Clearly, the fastest execution was observed in the case that used a 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 4.2). On the other hand, it is smaller than 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 possible if any two workflow tasks have a conflicting set of dependencies.

At the other end of the execution performance range 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. This 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 using a different task deployment option. Fig. 4.9 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

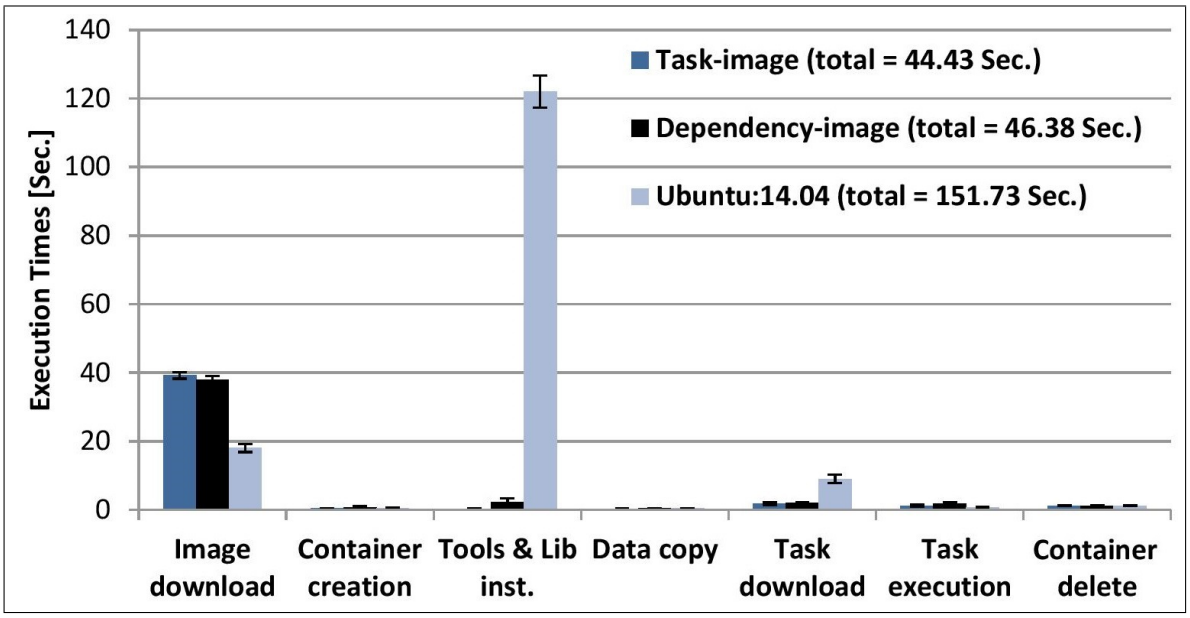

Figure 4.9: Execution time of the CSVExport task deployed using three task deployment options.

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 proves the flexibility required when task code changes frequently, and it ran almost as fast as the option that used a specific task image.

Importantly, the deployment options presented here can be mixed within a single workflow and can change over time as the workflow and tasks undergo changes in their development phases. We expect that for a newly created workflow the designer would use specialized task images for the common, mature tasks such as I/O transfer because they rarely change. In contrast, they would use on-demand installation for tasks specific to the workflow application. Then, once the iterative 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 production use, the designer can capture a workflow-specific Docker image with all tasks and dependencies preinstalled, which would then offer the best performance for the users.

4.6 Conclusions

In this chapter we presented a new system to build, deploy and enact scientific workflows. It integrates our TOSCA-based workflow definition presented in the previous chapter with container-based virtualization. The most important benefits of this approach is to improve the portability of the workflow, and the opportunity it creates for reuse. We used a small set of common life-cycle 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 approach 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 and provides a number of different deployment options. 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 approach 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 at the level of code distribution (using Docker images). Importantly, this approach is one of the key foundations of our approach to improve the reproducibility of scientific workflows, which is presented in the next chapter.

5

A Framework for Scientific Workflow Reproducibility in the Cloud

Summary

In the previous chapters, we showed how to describe the entire structure of a scientific workflow, together with all its components and a specification of the host environment. In this chapter, we can go further and turn workflows into reusable entities that include not only the description of a scientific experiment but also all details needed to deploy and execute them automatically.

We describe an approach that integrates our TOSCA-based workflow description, source control, container management and the automatic deployment approach presented in the previous chapter to facilitate workflow reproducibility. We have developed a framework that leverages this integration to support workflow execution, re-execution and reproducibility in the cloud, as well as in a personal computing environment.

We demonstrate the effectiveness of our approach by examining various aspects of repeatability and reproducibility for real scientific workflows. The framework allows workflow and task images to be captured automatically, which improves not only repeatability but also runtime performance. It also gives workflows portability across different cloud environments.

Finally, the framework can also track changes in the development of tasks and workflows to protect them from unintentional failures.

5.1 Introduction

Researchers in different disciplines have embraced workflows to conduct a wide range of analyses and scientific pipelines [40], mainly because a workflow can be considered as a model defining the structure of the computational tasks necessary for the management of a scientific process [78].

However, workflows are not only useful for representing and managing the computation but also as a way of sharing knowledge and experimental methods. When shared, they can help users to understand the overall experiment, or they can become an essential building block in new experiments. Lastly, workflows can also be used to repeat or reproduce the experiment and replicate the original results [63].

One of the major challenges in achieving workflow reproducibility, is the heterogeneity of workflow components which demand different, sometimes conflicting sets of dependencies. Ensuring successful reproducibility of workflows requires more than simply sharing their specifications. It also depends on the ability to isolate necessary and sufficient computational artifacts and preserve them with adequate description enabling future re-use [90].

A number of analyses and research efforts have already been conducted to determine the salient issues and challenges in workflow reproducibility [14, 48, 54, 145]. In short the issues can be summarized as: insufficient and non-portable description of a workflow including missing details of the processing tools and execution environment, unavailable execution environments, missing third party resources and data, and reliance on external dependencies, such as external web services, which add difficulty to reproducibility at a later time.

Currently, most of the approaches that address the reproducibility of scientific workflows have focused either on their *physical preservation*, in which a workflow is conserved by packaging all of its components, so an identical replica is created and can be reused; or on *logical preservation*, in which the workflow and its components are described with enough information for others to reproduce a similar workflow in the future [113].

Although both packaging and description, play a vital role in supporting workflow re-use, alone they are not sufficient to effectively maintain reproducibility. On the one hand physical preservation is limited to recreating the packaged components and resources but it lacks a structured description of the workflow. Therefore, it makes it easy to *repeat* exactly the same execution, yet it often does not enable the important ability to *reproduce* the experiment with different parameters or input data. On the other hand logical preservation can provide a detailed description of various levels of the workflow. However, this is still not enough if there is an absence of the necessary tools and dependencies required to execute workflow tasks. The need to integrate these two forms of preservation is therefore apparent.

In this chapter we present a new framework designed to address these challenges. The framework integrates features of both the logical and physical preservation approaches.

Firstly, we use our TOSCA-based approach that allows a workflow description to include the top-level structure of the abstract workflow, together with details about its execution environment. The description can offer portability in automated deployment across different execution environments including the Cloud and a local VM as presented in Chapter 4.

Secondly, using Docker virtualization our framework offers portable packaging of whole workflows and also their sub-parts. By integration with TOSCA, the packaging is automated, hence users are free from creating and managing Docker images. Additionally, our framework is built upon code repositories that natively support version control – crucial in tracking the evolution of workflows and their components over time.

We argue that combining these three elements: portable and comprehensive description, portable packaging and widely applied version control, play a fundamental role in maintaining reproducibility of scientific workflows over longer periods of time. They allowed us to build the framework which we present in this chapter. We evaluate the framework using real scientific workflows developed in our previous projects to demonstrate that it can effectively realise its goal.

The remainder of this chapter is organised as follows: Section 5.2 presents in details the requirements to improve the reproducibility of scientific workflows. Section 5.3 provides the description for the core and details of our framework for workflow reproducibility. In section 5.4, the structure and comprehensive description for repositories of workflows and tasks are presented. Then the evaluation of our solution is presented in section 5.5. Finally, Section 5.6 closes the chapter with conclusions.

5.2 Requirements for Workflow Reproducibility

A review of the literature shows that there is no standard approach to address the reproducibility of scientific workflows. However, there are several requirements and guidelines for achieving reproducibility that need to be addressed. In this section, we highlighted a set of these requirements, which are covered by the work presented in this chapter:

Workflows should be well described and annotated: Sufficiently detailed description and annotation of a workflow helps to define all tasks, properties and dependencies. Workflow description can play an important role into re-executing and reproducing the workflow, increase the comprehensibility of its purpose and functions.

A workflow's documentation should provide: 1) a structured description of the processing steps carried out in a workflow. 2) annotations for the functional aspect of the workflows providing the design and purpose of the workflow as well as the structure of the tasks involved in the workflow. 3) detailed descriptions of the tasks in a workflow to facilitate an understanding of what each task aims to achieve as well as their inputs and outputs.

Preservation and sufficient description of the execution environment: The execution environment is composed of the set of computational resources (software and hardware) that are involved in the execution of scientific workflow. The authors in [114] advised enriching the workflow description with information about the required execution environment. The execution environment can be preserved by adopting a physical approach, where actual resources including the operating system and its configuration are captured. However, this approach is not always viable due to frequent changes in the underlying computing systems (e.g.: changes in the operating system or its libraries). Therefore, it is important to collect appropriate details for the execution environment in such a way that will assist in the re-provisioning of an environment that can support the task, though not identical, and enables workflow re-execution.

Preserving and tracking changes of the external dependencies: many workflow tasks require specific software packages or libraries. Without these, a task execution will fail. For example, a task, relying on an R library, will fail if it is missing

or an incompatible version is used. Therefore, it is necessary to collect information about the software and library dependencies used to execute workflow tasks in order to maintain the ability to re-execute the workflow.

According to [145] and [87], volatility of these software and resources results in the decay of 50% of the tested workflows. Although changes in dependencies are not always controllable, providing sufficient documentation and regular change tracking enables the identification of the required version and different changes made to these resources. This will also help for selecting a specific library version compatible with workflow/task version as well as facilitating the process of choosing alternative library. Task dependencies can also be preserved by packaging them using available tools and techniques such as VMs or other containers.

Packaging sample data and auxiliary information: The analysis conducted in [145] reported that of the 92 workflows, 15% of them could not be re-executed because of the absence of sample data.

Missing input data might produce different results even when the inputs were described in detail, it might be difficult to create proper values for the input data to be used for workflow execution. Without input data, the ability to re-execute the workflow and understand its function is constrained. Therefore, providing sample data is helpful for the re-execution of the workflow.

Tracking changes and the evolution of workflows and tasks: Another factor that facilitates the reproducibility of workflow is versioning [141] [8]. By enabling version control on a workflow and all related components, a user can track the evolution of a workflow and its tasks, and retrieve a specific task version related to a workflow version. This allows the exact version of a workflow and its tasks used in the production of an experimental result to be used in the reproduction of that result.

To control changes that can affect a workflow and its components, we used a version control systems. This gives the ability to track the complete history of developmental changes of workflows and tasks.

Supporting workflow portability:

Having access to the detailed information about a workflow improves the ability to

reproduce it. However, it is important for reproducibility and repeatability that this description allows portability of the deployment across different execution environments [14]. This portability might be achieved by packaging a workflow using a portable mechanism such as Docker containers and images.

Sharing workflow, tasks and data: To reproduce an experiment represented by a scientific workflow, it is important that the data, code, and the workflow description are available [86]. A number of public repositories are available to share workflow specifications, task code, dependencies packages and tools and sample data [53] [5].

Offering a repository for the tasks in a deployable way provides the ability to store different versions of the task with different implementations, and all the software stack including operating system, libraries, and middleware. The ability to construct a repository of workflows/tasks that can be accessed and deployed during or prior to workflow deployment/enactment supports the following capabilities:

- Enabling developers to upload tasks/workflow to be stored in the repository so that they can be fetched later and used by other users.
- Allow access to specific versions of tasks from multiple workflow enactments so they can be reused in other workflows.

Automating workflow deployment: As stated in [71], another factor that contributes for better understanding and facilitate the re-usability of a workflow is the automation of workflow deployment. Therefore, providing workflow users with a mechanism for automatic workflow deployment has the potential to significantly improve workflow reproducibility.

5.3 Improving Workflow Reproducibility

The complete reproducibility of a workflow is hard to achieve due to possible changes at various levels in the software and hardware platforms used to run it. We can, however, significantly increase the degree of reproducibility by addressing the challenges discussed earlier: this was the goal behind the design of our workflow reproducibility framework.

5.3.1 The Framework Architecture

The proposed workflow reproducibility framework consists of four main components: the Core repository, a set of Workflow and Task repositories, the Image repository supported by Automatic Image Creation (AIC) (described in 5.4.4), and the workflow enactment engine (Fig. 5.1). The Core repository includes a set of common and reusable TOSCA elements such as Node- and RelationshipTypes, and life-cycle management scripts. They are a foundation for building tasks and workflows. The Workflow and Task repositories are used to store workflows and their components so they can be accessed during enactment and also shared and reused in designing new workflows. The Image repository contains workflow and task images that are used to improve both reproducibility and also workflow enactment performance. Images are automatically captured by the AIC. Finally, the workflow enactment engine is implemented by a TOSCA-compliant runtime environment.

To implement logical preservation we rely on the TOSCA specification which we previously adopted as a method to model portable workflows [107], as described in Chapter 3. With TOSCA we can describe workflows not only at the abstract level but also with the complete software stack required to deploy and enact them. This approach is portable because we have designed a way to use a TOSCA-compliant runtime environment to automatically deploy and enact our workflows on a range of different cloud platforms or in a local VM.

To control changes that can affect a workflow and its components we use a version control platform. This gives us the ability to track the complete history of developmental changes of workflows and tasks. The version control platform also supports

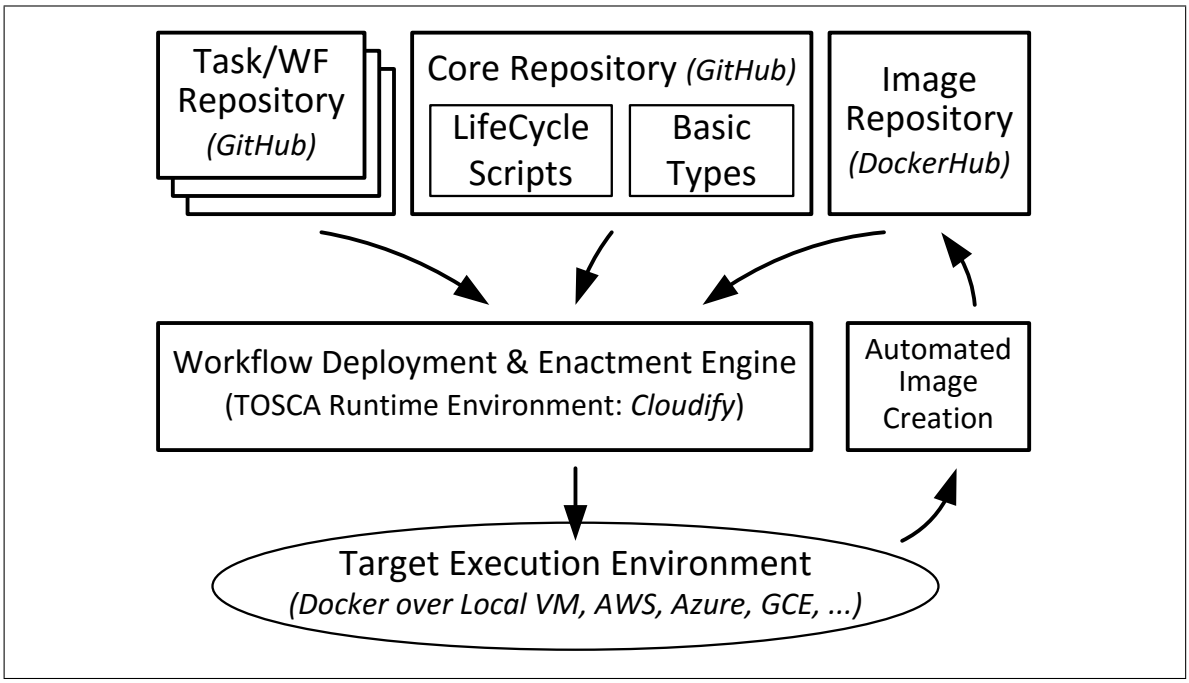

Figure 5.1: The architecture of our workflow reproducibility framework.

the Automatic Image Creation facility. The AIC uses Docker to implement physical preservation of workflows and greatly helps in the building and management of image libraries.

Instead of building yet another workflow repository and yet another workflow engine, we have defined our framework on top of open platforms such as GitHub and DockerHub. The former allows workflow and task source code to be stored and maintained under version control, the latter can store workflows and tasks packaged as Docker images. Importantly, both platforms offer mechanisms which promote sharing and reuse.

5.3.2 The Framework in Use

To create a workflow the user needs to follow our TOSCA-based modeling approach (described in Chapter 3) to implement and model its essential components including: NodeTypes and task code. The NodeTypes are used to declare tasks and dependency libraries, and they also refer to the task code – the actual software artifacts which will be deployed and executed.

To facilitate building new tasks and workflows we implement a set of basic NodeTypes

as shown in figure 3.3 along with tasks that others can reuse. Additionally, our Core repository provides RelationshipTypes and life-cycle management scripts (described in section 4.3.2) that are common to all workflows. They define and implement basic workflow functionality like passing data between tasks, the configuration of library dependencies, etc. Given all these components, the workflow can be encoded as a TOSCA Topology Template. The template (described in section 3.3.2) includes Node- and Relationship Templates that are instances of the types developed earlier; these templates represent tasks and task links, respectively.

Once the workflow Topology Template has been prepared, it can be deployed by a TOSCA-compliant runtime environment. The enactment of workflows follows the structure embedded in the Topology Template that in a declarative way combines components and dependencies. Using the Topology Template, the TOSCA runtime environment (Cloudify) can infer the appropriate workflow execution plan.

5.4 Workflow and Task Repositories

Since we have been using publicly available platforms like GitHub to maintain the Workflow and Task repositories, these repositories can remain under user control to clone, use and modify the workflows and tasks. We provide our own example repositories with a set of basic reusable workflow tasks and example workflows mainly to illustrate how the framework can support reproducibility. But primarily, the ecosystem of workflows and tasks will be grown by researchers and scientists who want to develop their own workflow applications.

The choice of source version control platforms, such as GitHub, to host repositories of workflows and tasks was not accidental. These platforms offer excellent tools to support sharing and communication. But more importantly, they allow code developers and users to keep track of the changes as code is developed and this can directly help to improve repeatability and reproducibility.

Our approach works on the principle that each single workflow and workflow task is maintained in a separate code repository. That brings multiple benefits: repositories mark clear boundaries between components, they offer independent version control,

allow for easy referencing and sharing, and additionally, provide branches and tags to implement strict control of workflow and task interface. With multiple repositories it's also easy to encapsulate auxiliary information, such as sample data and human readable description specific to each workflow and task, which helps to maintain long-term reproducibility.

5.4.1 Repository Structure

A repository aggregates various artifacts with information and resources related to the workflow or task. These artifacts as shown in fig 5.2 include: TOSCA-based descriptors, workflow/task-specific life-cycle scripts, sample data, human readable description, the *one-click* deployment script and deployment instructions, input.yaml file for workflow deployment parameters. The mandatory - and most important artifacts are TOSCA-based descriptors. In the case of a workflow, this is the ToplogyTemplate descriptor described in Chapter 3 (see Appendix A for a sample template), that encodes the structure of a workflow and references all the workflow components and life-cycle scripts required for enactment. In the case of a task, the descriptor includes the TOSCA NodeType (described in section 3.4.1) that defines the task interface and refers to the actual task implementation code in addition to a sample TopologyTemplate includes the NodeTemplate task. The input.yaml is a yaml-based file that provides number of setup parameters for workflow deployment used during the deployment process such as: user specified Docker image that will be used to create workflow/task container, the location and name of the sample data file and a flag variable used to indicate whether the user want to create Docker images or not.

Other artifacts, although optional, are helpful for maintaining reproducibility. For example, when provided with sample data and the one-click deployment script (described in section 4.3) users can easily test a workflow or task in their environment. The deployment script starts a multi-step process which deploys the workflow together with basic dependencies such as Docker and Cloudify and then enacts it. The human readable description, shown in figure 5.3, includes: description about workflow function, inputs, outputs, information about tools required to deploy the workflow (Cloudify and Docker) along with execution environment specifications. This description is stored

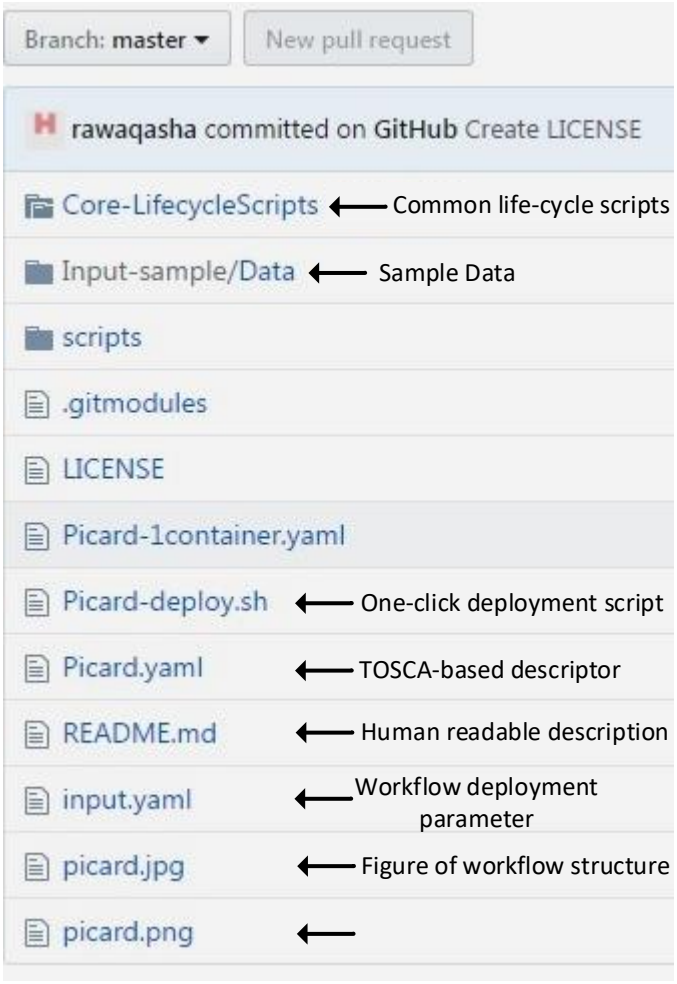

Figure 5.2: The artifacts of a workflow repository.

in a repository and enables users to understand better the purpose of the component and so use it more easily. This can also help users to recover from failures in the face of changes in the workflow or any of its dependencies.

The structure of workflow and task repositories are very similar. This is because our tasks also include a simple test workflow descriptor and sample data which allow users to easily run a task and test whether it actually meets their requirements. Maintaining a separate repository for each task of a workflow facilitates the understanding, maintenance, re-use and separate testing of the workflow.

Usually, our repositories include two workflow descriptors that define the *single-* and *multi-container* configurations to support different deployment scenarios. As discussed

in the previous chapter, the single-container workflows are executed within one Docker container, whereas in the multi-container configuration *each* task runs in its own container. The use of the single- or multi-container configuration also has an impact on the kind of images that will be generated by our Automatic Image Capture facility.

These two default configurations so however only describe two extremes out of the range of possible workflow deployments. For more specific, advanced scenarios developers can create workflows that include containers which group together a subset of tasks, for example for sharing the same dependencies.

5.4.2 Interface Control via Branches and Tags

One of the major sources of workflow decay is changes in the components that a workflow is comprised of. In a living software system changes are inevitable because the components – tasks, libraries and other workflows – undergo continuous development. Yet to maintain reproducibility we cannot forbid all changes. Instead, we need to control them, so they do not contribute to the decay.

The changes that occur naturally during workflow and task development can affect two layers of the system: the interface and/or implementation of a component. By the workflow/task interface we consider the contract between the developer and user of a component. Specifically, it is the number and type of input data items and properties that the workflow/task uses in processing and also the number and type of output data items it produces.

Changes in the interface, such as adding a new input parameter, usually indicate some important modification to a component and need to be followed by changes in its implementation. Conversely, changes to the implementation only, if made carefully, are often merely improvements in the code which have no implications for other parts of the workflow.

Since, in our framework each component is maintained in a separate repository, we can control these two types of changes effectively. Figure 5.11 depicts an example of change tracking for the *SC* workflow. We use repository branches to denote changes in the interface such as adding new input parameter to a task or changing the type of

The Sequence Cleaning workflow (SC) is one of the steps in the Next Generation Sequencing pipeline.

WF-Title: The Sequence Cleaning workflow (SC)
version: 1.0
Description: The workflow is one of the steps in the Next Generation Sequencing pipeline. It was designed

in the e-Science Central system.

###WF-Tasks:

No-of-tasks: 8
Tasks: {importDir: 1, Pick-File: 1, Picard-Clean: 1, Picard-Mark: 1, Picard-Add: 1, SAMTools-index: 1, ExportFiles: 2}
Dependency-Libs: {java1.7: all, SAMTools-lib: SAMTools-index}

###Blueprint:

blueprint-name: Picard.yaml
Docker-images: dtdwd/picard1
sizes: 268 MB (Virtual size 594.6 MB)
OS-types: ubuntu14.4
tools: Java1.7, SAMTools-lib

###Input:

input-Dir: {
Probe.PFC_0030_MSt_GAGTGG.sorted.bam.bai, Probe.PFC_0030_MSt_GAGTGG.sorted.bam,
Probe.PFC_0030_MSt_GAGTGG_nodups.sorted.realigned.Recal.bai,
Probe.PFC_0030_MSt_GAGTGG_nodups.sorted.realigned.Recal.bam,
Probe.PFC_0030_MSt_GAGTGG_nodups.sorted.realigned.Recal.reducedReads.bai,
Probe.PFC_0030_MSt_GAGTGG_nodups.sorted.realigned.Recal.reducedReads.bam
}
description: input Dir including 5 input files

###Outputs:

output-folder: '~/blueprint-name'
output-file(s): {index-BAI-files, output-SAM_BAM-files}
description:
types: {' ', ' '}

###Execution-Environment:

Cloudify-version: 3.2
Docker-version: 1.8+
OS-type: ubuntu14.04
Disk-space: 10 GB
RAM: 3 GB

Figure 5.3: The human readable description of a workflow repository presented in README.md file as shown in 5.2.

task output (represented by orange and green lines respectively), and tags to indicate significant improvements in the implementation such as releasing a new version of a workflow, task or dependency library (represented by red circles). Minor implementation changes are simple commit events in the repository which do not need any special attention (represented by grey circles in the graph). This approach supported by the effective way that GitHub allows references to a specific branch or tag is enough to address the problem of changing components.

However, these mechanisms are not only important for allowing our framework to maintain the reproducibility of existing workflows, they are also crucial for users creating new workflows. With repository branches, users can easily see different flavours of a specific task or workflow and decide which one to use. On the other hand tags help users to see major improvements of a component or workflow. Tags also indicate to our framework when there is a need to create a new component image.

To illustrate the use of branching and tagging in practice we show later, in the Evaluation section, a development scenario for one of our test workflows.

5.4.3 Automatic Workflow Deployment

The model of describing workflows using TOSCA proposed in Chapter 3 is important because it not only supports logical preservation but also offers the ability to automatically deploy and enact our workflows. That facilitates repeatability and improves workflow reproducibility.

To run a workflow using our framework, users need to clone its repository to a target machine in which they are going to run it. The repository includes sample data and the *one-click* deployment script. This is a simple script able to install the software stack required to run the workflow (Cloudify, Docker and some auxiliary tools) and then to submit the workflow to Cloudify with default configuration parameters.

The default configuration and sample data make it easy for users to test the workflow. It also provides a way to repeat the execution as well as creating a starting point for reproducing it. To repeat a workflow users can simply switch to a very specific version of the workflow in the repository history and run the *one-click* deployment

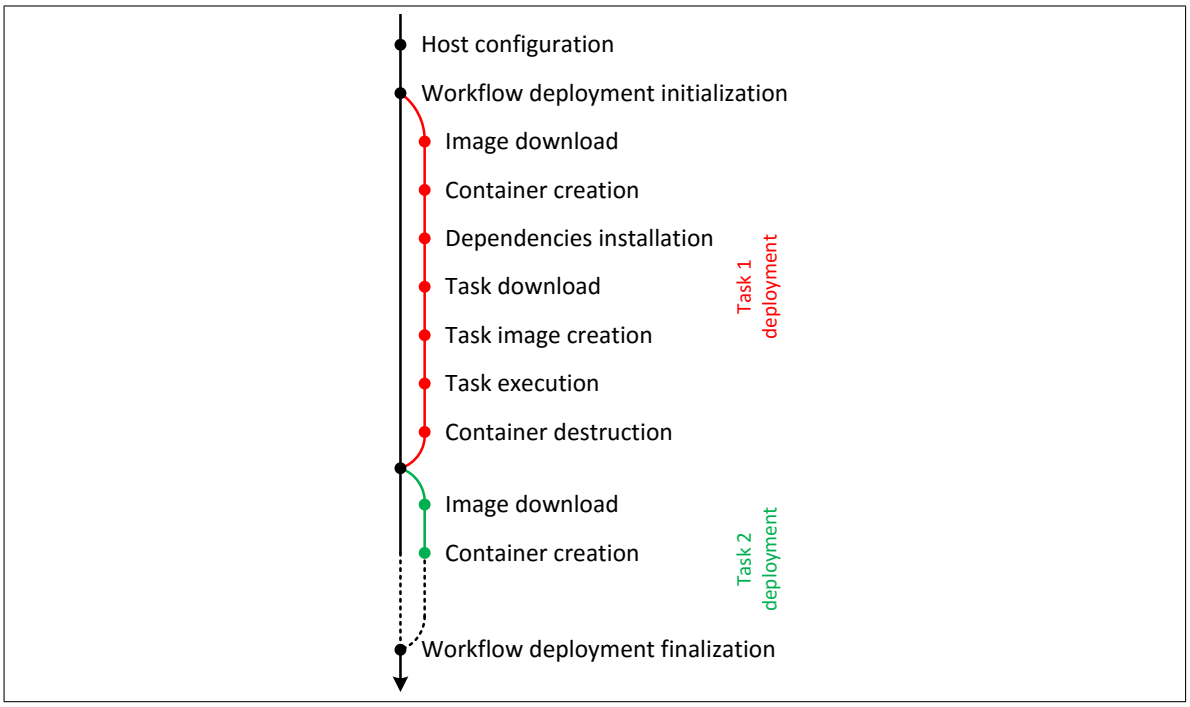

Figure 5.4: Steps in automatic workflow deployment using the multi-container configuration.

script. Then, they can modify the default configuration and provide their own data. For example, the default configuration refer to the attached sample data folder and files, but a user can change this to refer to their local data artifacts. They can also switch to the latest version of the workflow - the last tag of workflow repository - to validate the output or compare it with output generated by previous versions.

The TOSCA descriptor of a workflow is a declarative specification that includes all tasks, dependency libraries and task links embedded in the workflow ServiceTemplate. The template also includes dependencies against the task execution environment which may be composed of one or more Docker containers and VMs. Apart from the declaration of tasks and libraries, the workflow ServiceTemplate also encodes the topology of the workflow. For scientific workflows, usually implemented as directed acyclic graphs, there is sufficient information to allow a linear workflow execution plan to be automatically inferred (Fig. 5.4). Cloudify follows the generated plan, and deploys and runs one task at a time.

Crucial to workflow enactment are life-cycle management scripts. As described in the previous chapter, they implement deployment operations that each workflow and task needs to go through, such as: initialization of a shared space used to exchange

data between tasks, provisioning of the host environment (a container), installation and configuration of library dependencies. As the majority of tasks follow a very similar pattern of deployment, we developed a set of common, reusable life-cycle scripts (described in section 4.3.2) and included them in the Core repository. Developers can refer to these scripts when building their own workflows and tasks.

5.4.4 Automatic Workflow/Task Image Capture (AIC)

TOSCA-based descriptors are the fundamental element of our framework, partly because they are used to implement logical preservation (described earlier), and partly because they allow workflows to be automatically deployed and enacted. However, running workflows based only on these descriptors would result in significant runtime overheads. The framework would repeat the same, sometimes long running, steps to deploy a task every time it was executed, both if re-used in the same workflow, or in a range of workflows.

As mentioned in Chapter 4, our deployment approach is flexible enough to run the workflow and task deployment process using a variety of Docker images:

1. starting from a pure OS image - typically one available from DockerHub
2. using a specific user-defined image which includes some workflow/task dependencies
3. using a complete image that contains all of the required dependencies.

If the image referred to in the workflow ServiceTemplate does not contain all the dependencies (as in 1 and 2 above), they will be installed by the framework on-demand during workflow enactment. This automation simplifies the development cycle because users are not forced to manually prepare and manage task or workflow images before they can use a workflow.

Yet, to simplify the use of the framework even further we implemented an Automatic Image Capture facility. As described in Chapter 4, a container is created during the provisioning of each workflow task in order to host the task and all the required dependencies. Following the provisioning process of all dependencies and downloading the

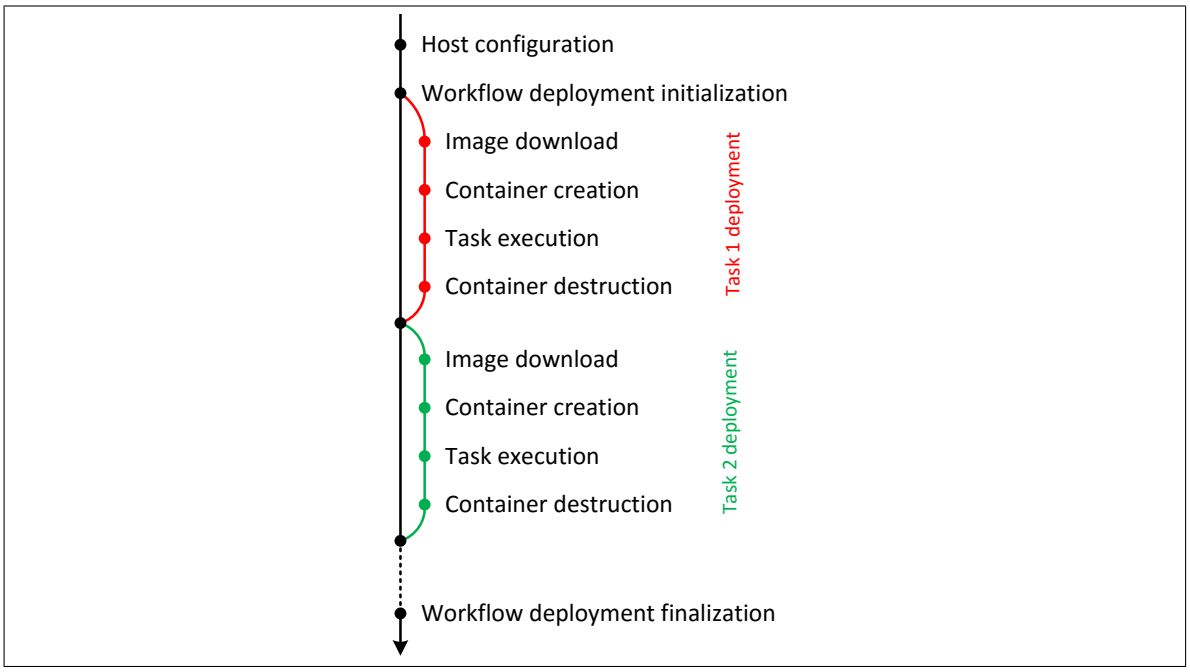

Figure 5.5: Steps in automatic workflow deployment using the task images created by the AIC; cf. Fig. 5.4.

task, the AIC facility automatically creates a Docker image using the task container. The created image is given a unique name that includes the task name and the version (represented by the branch and tag of the task repository).

The AIC automatically creates these images for both the workflow and its task, so that they can be deposited in a private or public Image Repository. The next time a task is executed, instead of going through the complete deployment cycle, the framework will automatically use these images created earlier by the AIC (Fig. 5.5). As shown in the Evaluation section 5.5.2 below, this automatic optimization to the deployment process can have very significant impact on runtime performance.

The workflows we implemented are usually described with two configuration options: single- and multi-container. These options influence the way in which deployment and enactment of workflows is performed, but also determine what image the AIC will create for the workflow. If the workflow uses the single-container configuration, the AIC will capture a single image that encapulates the whole workflow including all its components. Conversely, if the workflow uses the multi-container configuration, one (smaller) image will be created for each task. Both options have their advantages: the former imposes less overhead in terms of storage and performance, whereas the latter

promotes better reuse of task images and gives more flexibility if the workflow requires updates. Nonetheless, they support the repeatability and reproducibility of workflows equally well.

However, to realise this goal, images must be properly versioned. The AIC uses identifiers from the Image Repository and tags from the Workflow and Task Repositories to achieve this. The workflow/task image identifier is generated based on the base Docker image identifier and the URL of a branch or tag of a workflow/task for which the image is built. This simple and *unique* mapping between code and image versions allows users to include only the *code* URL in their workflow ServiceTemplate. This is enough for the framework to fetch and use the correct image for a task or workflow. And in the case that the image does not yet exist, the workflow enactment will follow the full deployment cycle during which the AIC will automatically generate and deposit the relevant images for future use.

5.5 Evaluation and Discussion

We describe the evaluation of our framework from three different perspectives. First, we present a set of experiments to show the portability of the workflow description - the fact that it can be enacted in different environments. Second, we show the benefit of using the AIC to reduce a workflow's runtime. Thirdly, we describe a specific scenario including both workflow and task development to illustrate how the framework can maintain reproducibility in the face of changes in workflow components.

5.5.1 Repeatability on Different Clouds

The goal of this set of experiments was to re-enact a workflow, initially designed in a local development environment, on three different Clouds as well as a local VM. We ran the experiment for the four different workflows described in Chapter 4. The workflows: *Neighbor Joining* (NJ) (figure 3.4), *Sequence Cleaning* (SC) which is used in a NGS pipeline [29], *Column Invert* (CI) and *File Zip* (FZ) are different in terms of structure, the dependency libraries they require and the number of tasks they include (11, 8, 7 and 3 tasks, respectively). Figures 5.6, 5.7 and 5.8 depict the structure of

the Sequence Cleaning, Column Invert and File Zip respectively according to their TOSCA description.

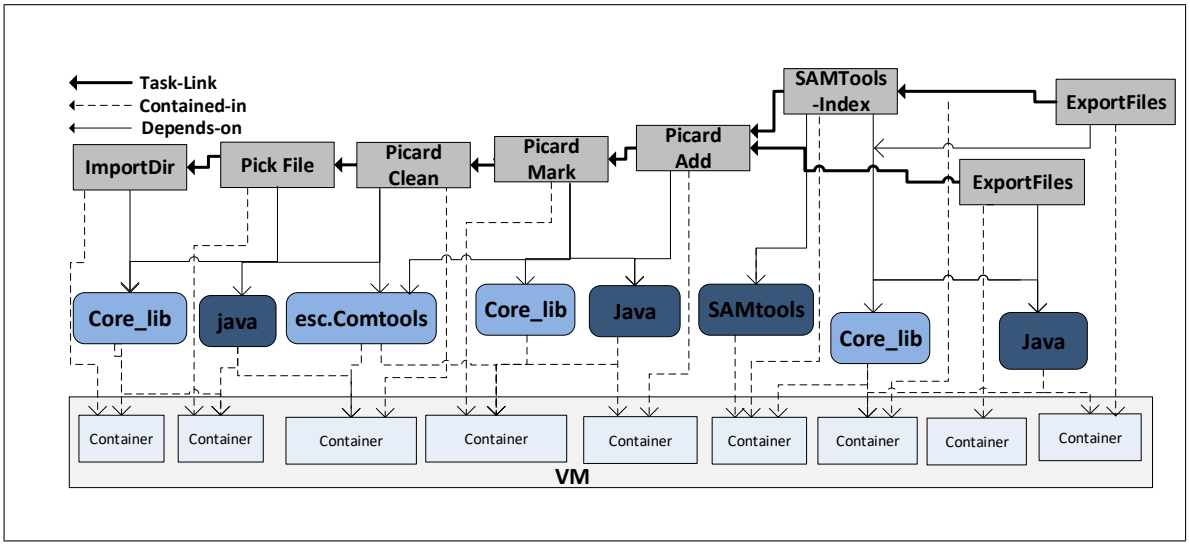

Figure 5.6: The structure of the Sequence Cleaning workflow in multi-container configuration described in TOSCA.

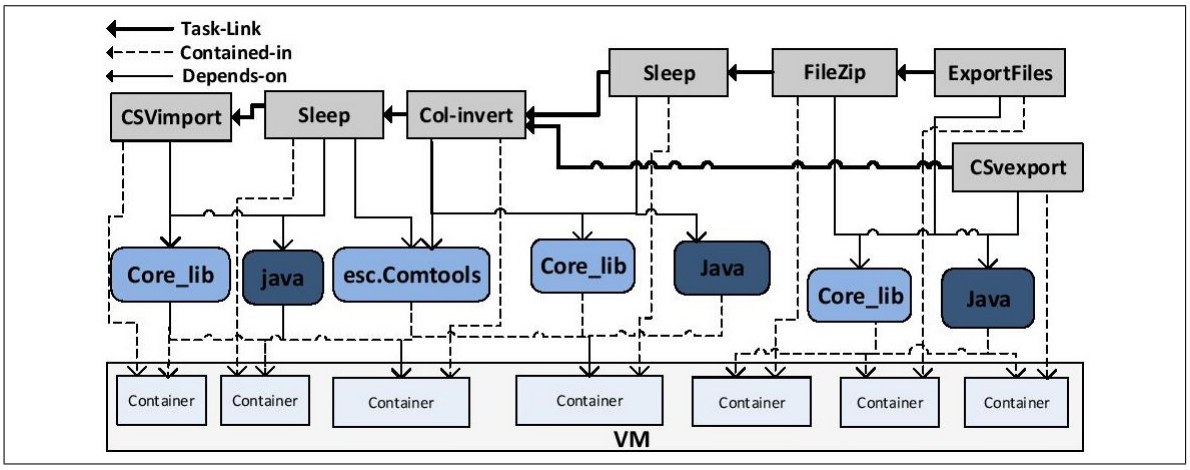

Figure 5.7: The structure of the Column Invert workflow in multi-container configuration described in TOSCA.

To illustrate the potential of our framework in supporting repeatability and reproducibility and the value of the proposed workflow representation, each of the selected workflows was first developed, and then deployed and enacted in a local development environment. We recorded the execution time for that initial enactment which included automatically creating Docker images for the whole workflow or individual tasks (depending on the desired deployment scenario).

To conduct the rest of the experiment, we cloned the workflow repositories in four

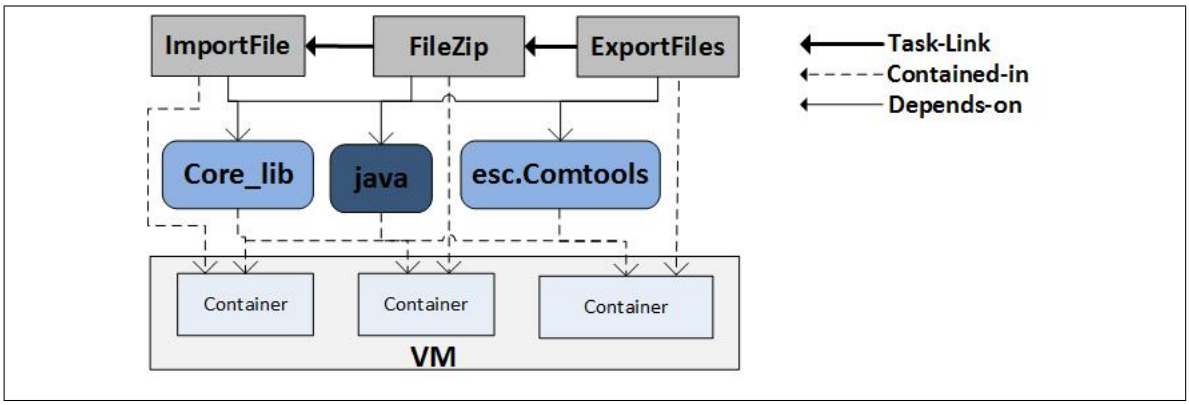

Figure 5.8: The structure of the File Zip workflow in multi-container configuration described in TOSCA.

Table 5.1: Basic details about the execution environments.

Environment	CPU Cores	RAM [GiB]	Disk space [GB]	Operating System
Local VM	1	3	13	Ubuntu 14.04
Amazon EC2	1	1	8	Ubuntu Srv 14.04
Google Cloud	1	3.75	10	Ubuntu Srv 14.04
Microsoft Azure	1	3.5	7	Ubuntu Srv 14.04

different environments: a local VM, and Amazon AWS, Google Engine and Microsoft Azure Clouds. Finally, we re-executed workflows five times in each environment using the pre-built images and collected the results. The configuration of the VMs is presented in Table 5.1.

Each workflow was used in two different configurations - single- and multi-container - to show the overheads of running multiple task containers. The input and output data for the workflows were the same in all executions and the average execution times were similar. Fig. 5.9 shows a chart with the results for the SC workflow and Table 5.2 includes the results for the other workflows.

The experimental results show that our scientific workflows can be re-enacted, producing the same outputs with a similar runtime. They illustrate a common development pattern in which developers build and test a workflow in their local environment and once it is ready they can share it with others via Workflow, Task and Image Repositories. The integration of TOSCA representation and Docker packaging offers significant support for this pattern.

Table 5.2: The average execution time (in minutes) for different workflows executed in different environments.

	Neighbour Join.		Column Invert		File Zip	
	Single	Multi	Single	Multi	Single	Multi
Development Env.	2.13	2.54	0.9	1.3	0.6	0.94
Amazon	1.74	2.27	0.66	1.18	0.5	0.84
Azure	2.52	3.86	1.35	2.1	1.23	1.38
Google	1.52	2.48	0.74	1.18	0.5	1.01
Local VM	1.65	2.5	1.03	1.37	0.53	1.03

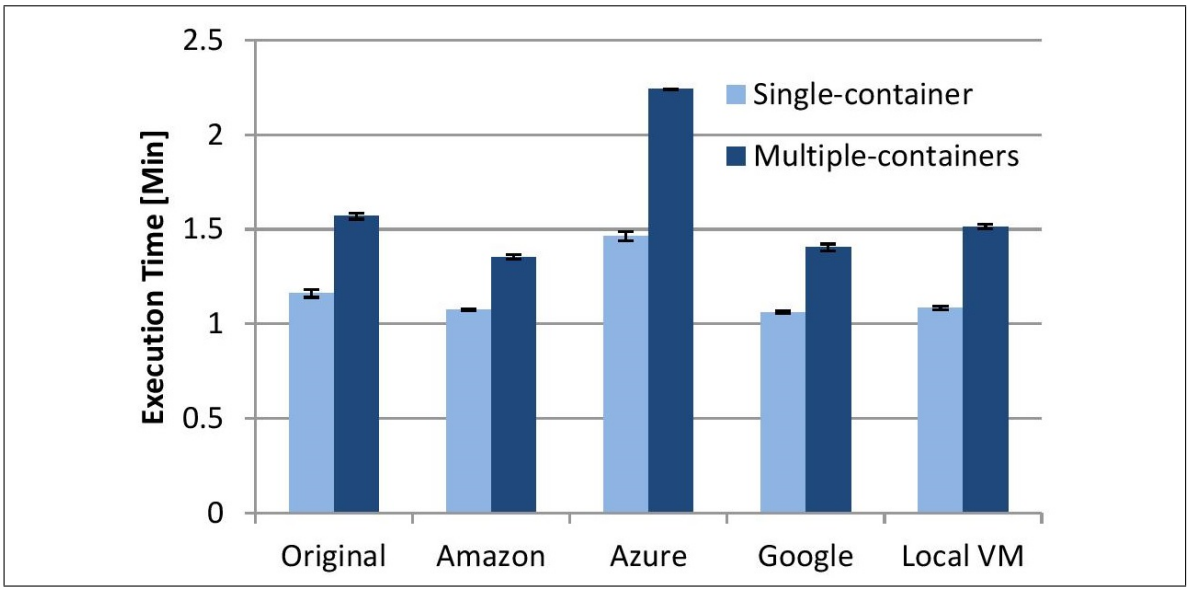

Figure 5.9: The average execution time for the Sequence Cleaning workflow executed in different environments.

5.5.2 Automatic Image Capture for Improved Performance

As mentioned earlier, our framework is flexible enough to allow tasks and workflows to use pure OS images available from DockerHub or custom, predefined task/workflow images created by users or the AIC. By using a predefined image we can avoid the installation of dependency libraries and task artifacts required during workflow execution. As shown previously in Fig. 5.5, this can reduce the number of deployment steps required in workflow enactment.

The elimination of some of the deployment tasks can have very positive impact on the runtime of workflows. To show this, we prepared a set of experiments in which we

ran our workflows using different images: the base image available on DockerHub, the base image with pre-installed dependency libraries and task images captured by the AIC. Fig. 5.10 depicts the average workflow execution time for four workflows.

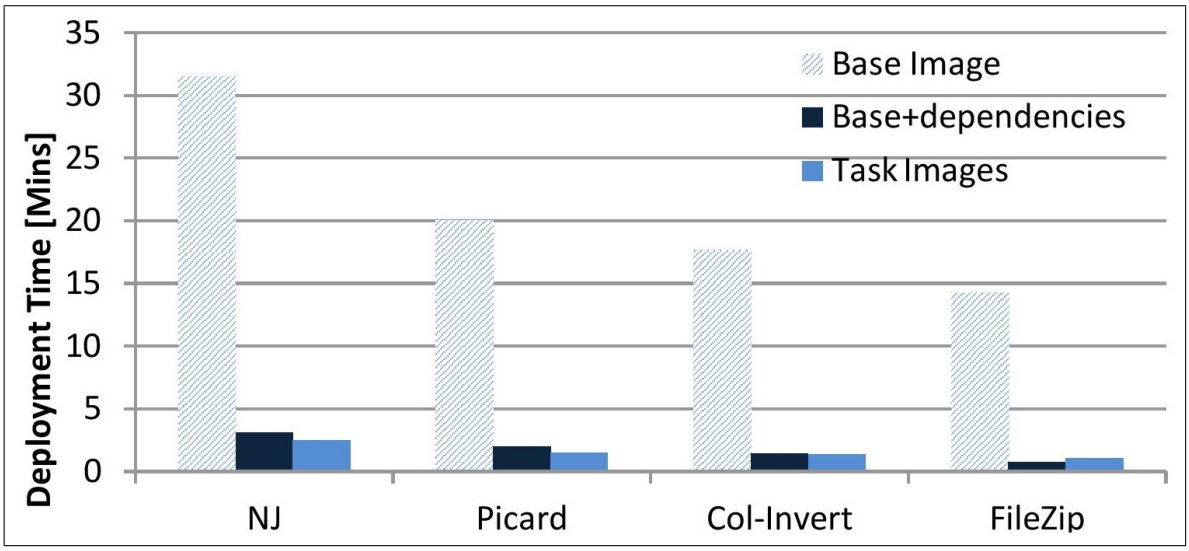

Figure 5.10: The average execution time of test workflows using different task images.

Clearly, there was a significant overhead in using the base image from DockerHub which included automatically creating Docker images for each individual tasks. The main reason was the time required to install dependency libraries such as the Java Runtime Environment or, in the case of the NJ workflow, the Wine library.

The second and third option show small differences with slightly shorter execution for experiments which used images created by the AIC. That is because the AIC captures everything the task needs to run (according to the task's TOSCA descriptor), whereas the second option included only dependency libraries while the task artifacts (task code) were downloaded and installed on-demand during every execution.

The results show that from a performance perspective the use of pre-packaged images is the most effective option. However, from the user perspective, the quickest and easiest is the use of the base images already available on DockerHub instead of building images manually. Our framework supports the flexibility of building the images automatically to avoid the need for the users to build them manually for the cost of some overhead incurred by the initial execution of a workflow. The first run will involve the complete deployment cycle and creation of the images, whereas any subsequent executions will benefit from those images and will run at full speed.

5.5.3 Reproducibility in the Face of Development Changes

One of the key factors that can reduce the decay of our workflows is their ability to embrace changes that occur naturally during workflow and task development. These changes can mainly affect two layers: the input/output interface of a workflow or task, and their implementation.

In Fig. 5.11 we illustrate a *realistic* evolution scenario of the Sequence Cleaning workflow shown earlier in Fig. 5.6.

The left side depicts the timeline of development events that occurred during the scenario. It is accompanied by change trees from two repositories: the left tree represents the evolution of the workflow, the one on the right shows the evolution of one of the workflow tasks.

We start the analysis with the version of the SC workflow presented earlier and tagged as `v1` in Fig. 5.11 (event 1). By tagging, we acknowledge that this version has been published, advertised and so may be used by others.

Now, let us imagine that a new requirement for our workflow appeared (event 2) – users of the workflow want to save storage space by compressing the workflow output files. In response to that, the developers created a new `Zip` task (cf. the right version tree) and wanted to add it to the workflow. Note however, that changing the type of outputs generated by the workflow is a change of its interface. This would likely break any external application that has used uncompressed outputs provided by version `v1`. Thus, before we can add the `Zip` task to the workflow we need to create a new branch, named `zipped` in the figure (event 3).

The `zipped` branch of the workflow refers to the `Zip/master` branch of the task. By default such a reference means that the workflow depends on the latest tagged version of the task coming from that branch. This is convenient because as the task implementation is improved over time, the `zipped` workflow will use a task's latest tagged version (including `v1.1`). In this way workflows are updated automatically without the need to change them when only implementation improvements are made to the tasks. However, if strict workflow repeatability is required, the reference to the `Zip` task would include a specific tag. That would prevent the automatic update of

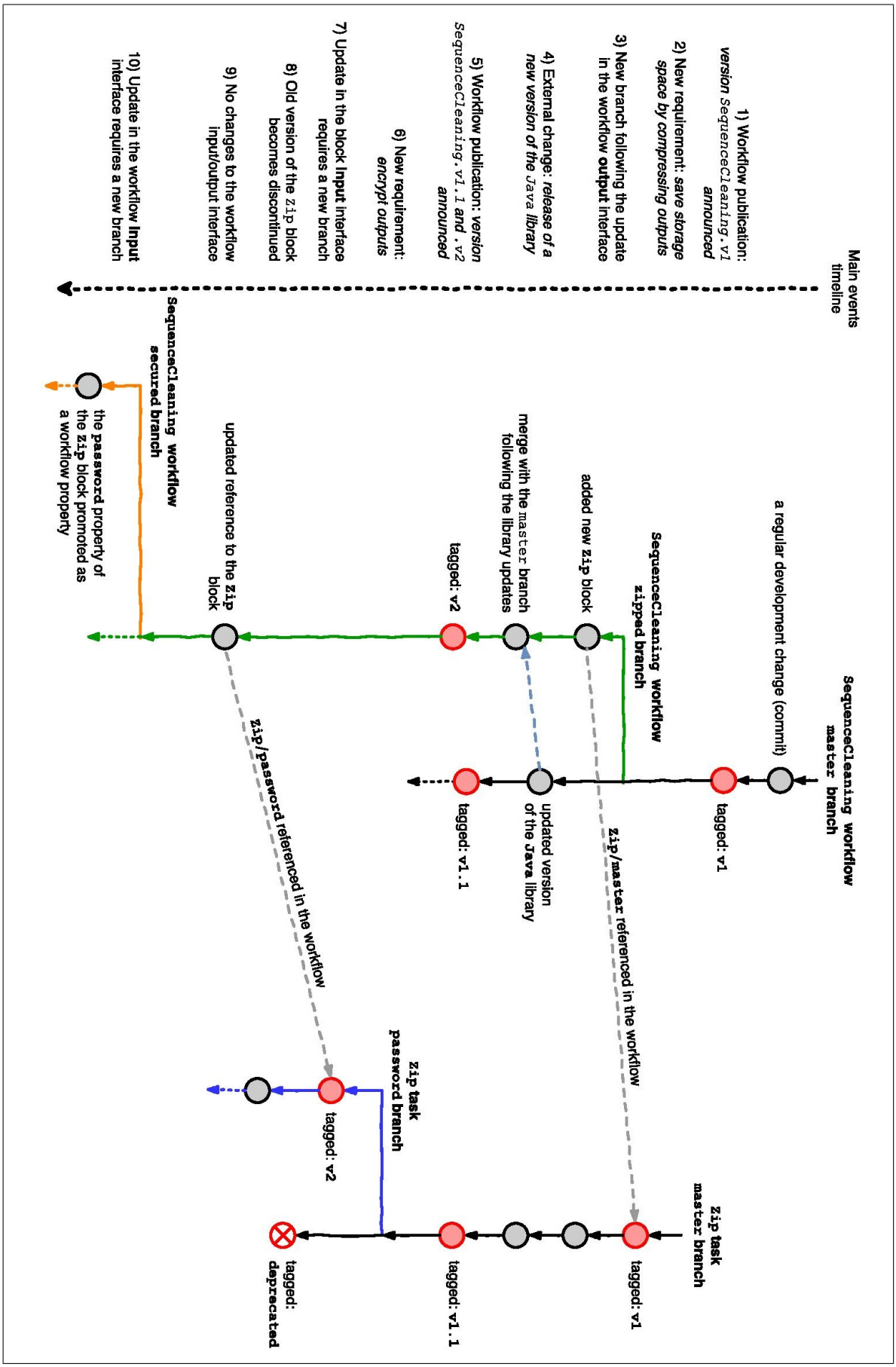

Figure 5.11: A hypothetical evolution of the `Sequence Cleaning` workflow.

such a workflow.

Next, event (4) denotes a new release of the `Java` library used by some tasks in the workflow. In our hypothetical scenario the new version of the library has improved performance, and several errors have been fixed. Thus, the event is a signal for us to update the workflow as soon as possible. That change is compatible with the previous version of the workflow and so we do not need to create a new branch. Instead, we merge in the changes from `master` to the `zipped` branch, so that both branches can benefit from the updated library.

After adding the `Zip` task and updating the `Java` library, we also tag and advertise new, improved versions of our workflow (event 5). Specifically, `SampleCleaning/v2` runs faster and produces smaller outputs, which is an advantage to users.

Event (6) marks the arrival of yet another requirement – users want the outputs of the workflow to be encrypted to avoid leakage of patients' raw genomic data. That however requires some improvements in the `Zip` task, including changes to the underlying tool used to compress the data.

After running some tests it was clear that the new zip tool had much better performance, and so we quickly decided to swap the old implementation with the new tool and tag the task `v1.1`. Note that this simple act of tagging a version causes an automatic update of all workflows that rely on that branch. Therefore, from now on the `SampleCleaning.v1.1` and `.v2` workflows will use the updated implementation of the `Zip` task.

Continuing with the task update, we create a new `password` branch in the task repository (event 7). This new branch is needed due to the changes in the task's input interface – the new version has the extra `password` input property. But the use of encryption is optional, so to limit the number of branches we decided to discontinue the previous version of the `Zip` task and tag the branch `master` as `deprecated` (event 8). That indicates to users that they should use other branches of the task in their new workflows. Nonetheless, the old version will need to remain in the repository because others may still use workflow `SampleCleaning.v2` which relies on the `Zip/master` branch.

As proactive workflow developers we noticed that the `master` branch of the `Zip` block has been deprecated and so decided to update the reference in the `zipped` version of the workflow to the active `password` branch. Note that this update does not require a new branch because the use of encryption in the `Zip` task is optional. Thus, the workflow's input and output interface can remain the same (event 9).

The new branch is created later (event 10) when the `password` property is exposed to end users as the workflow input property. We want the users to be able to set a custom password for the output data and that requires a change in the workflow interface which, in turn, requires a new branch.

The hypothetical evolution we have presented shows a very common patterns in the development of workflows and their components, with changes occurring at different layers of the workflow and its tasks. However, due to having separate task and workflow repositories, combined with the tagging and branching of the code, we can maintain all workflow versions in a working state and ensure that their evolution does not break external applications that rely on them.

5.6 Conclusions

Reproducibility is a crucial requirement for scientific experiments, enabling them to be verified, shared and further developed. Therefore, workflow reproducibility should be an important requirement in e-Science. In this chapter we presented the design and implementation of a framework that supports repeatability and reproducibility of scientific workflows. It combines both logical and physical preservation. To implement logical preservation we use our TOSCA-based modeling approach as a means to describe workflows in a standardised way. To realise physical preservation we use lightweight virtualization which allows us to package workflows, tasks and all their dependencies as Docker images.

Our framework is unique in combing software repositories to manage versioning of source code, an automated workflow deployment tool that facilitates workflow enactment and reuse, and automatic image creation to improve performance. They all significantly increase the degree of workflow reproducibility. And although, our frame-

work does not capture *retrospective* provenance traces, (this has been left for future work), the proposed TOSCA-based workflow descriptors may be considered to be a detailed *prospective* provenance document. They describe the high-level structure of the workflow, and might also be encoded using, for example, the ProvONE specification,[1] together with all details needed to recreate the complete software stack needed for deployment and enactment.

[1]The latest draft of the ProvONE specification from May 2016 is available at: http://jenkins-1.dataone.org/jenkins/view/Documentation%20Projects/job/ProvONE-Documentation-trunk/ws/provenance/ProvONE/v1/provone.html

6

New Techniques for the Optimization of Scientific Workflow Deployment in the Cloud

Summary

Previous chapters have focussed on supporting portable modeling for scientific workflows that can be utilized to automatically deploy the workflows. As a consequence, our automatic deployment approach has been used to improve workflow reproducibility which is one of the significant benefits of our framework, which was presented in Chapter 5.

In order to improve the effectiveness of our reproducibility framework, in this chapter we describe and evaluate new optimization techniques that are shown to significantly optimize the performance of our deployment approach. Our work concentrates on the sharing and re-use of ready-to-run workflows and tasks that have been packaged as images. We propose a new algorithm to name and select compatible task images, which we integrated with a version control system. That allowed us to automate image creation, caching and then sharing. The effectiveness of the proposed techniques is evaluated via various scenarios in which we run real and synthetic scientific workflows in local and cloud environments.

6.1 Introduction

Scientific workflows play a vital role in modern science as they enable scientists to specify, share and reuse computational experiments [40]. Yet workflows need to be reproducible in order to maximise the benefits they provide and facilitate the sharing of knowledge about experimental methods.Workflow reproducibility enables effective sharing as scientists can re-execute experiments developed by others and more quickly create new and/or improved experiments [8].

A lot of work has been carried out to enable the sharing of workflows. Some rely on packaging workflows (physical preservation) and/or their components and sharing the packages so they can be re-used. Others focus on sharing the abstract description of the workflow (logical preservation) and task coordination. Both approaches rely on workflow creators publishing workflows and their components in addition to manually selecting, installing and configuring all the shared tools required to enact a workflow.

Much less attention has however been devoted to the automation of the deployment of workflows, their components and dependencies. Yet, by addressing that issue we are able to deliver highly reproducible workflows. Our approach, presented in Chapter 4, provides the automated deployment of workflows by uniquely combining the TOSCA specification and Docker technology. Our deployment approach is an effective part of the reproducibility framework which offers ready-to-run workflows and tasks, and addresses the majority of issues related to workflow decay and reproducibility.

However the delivery of highly reproducible workflows may come at the price of additional overheads. We observed some performance issues that impact the packaging of workflow components, their provisioning and, ultimately, workflow enactment. Specifically, the deployment of workflows may become slow if task images are pulled from remote repositories too often or if the same task or dependency is repeatedly provisioned. In this chapter we address these challenges by introducing image and cache management mechanisms that are shown to greatly improve the performance of both the provisioning and enactment of our reproducible workflows. As a result, not only do workflows enact more quickly when run in isolation, but we observe additional speed-up when they are executed concurrently on the same host or in parallel on different hosts (including clouds). This occurs often in real situations when a workflow is run repeatedly on different input datasets (for example, datasets streaming in form of sensors or applications). The proposed image and cache management is also integrated with source version control of the workflows and tasks. This allows backward-compatible changes to the task code to be transparently distributed on-demand across all workflow enactment engines. Thus, the proposed solution can help developers to streamline the process of building and distributing their workflows and tasks.

In this chapter, we present the following contributions: (1) a new algorithm to name, create and select a compatible task image that improves the re-usability of ready-to-run workflow components, (2) a multi-level cache of deployable components that supports workflow sharing and optimizes the workflow deployment process, (3) caching workflow components such as task artifacts and dependency packages to support the process of image creation, and (4) a set of experiments that use real and synthetic scientific workflows running on local and cloud environments to validate and evaluate

the proposed mechanisms.

The rest of this chapter is structured as follows. In Section 6.2 we give an overview of our proposed optimization techniques. Section 6.3 presents details of the image and cache management and is followed by evaluation through experiments discussed in section 6.4. We then draw conclusions in section 6.5.

6.2 Performance Optimization for Automatic Deployment

A scientific workflow typically consists of a set of components, each representing a scientific task that are logically connected to define a specific data flow.

However, with the rising performance demands placed on workflow execution due to the increase in ”big data” analytics application, it becomes increasingly important to optimize the provisioning of scientific workflows. To address this issue, we designed new optimization techniques and added them into our scientific workflow reproducibility framework.

6.2.1 Dynamic Workflow Deployment

One of the core functionalities of our reproducibility framework for scientific workflow is the on-demand, automatic deployment approach [109] presented in Chapter 4, using a TOSCA-based description and Docker technology.

In this approach, as described in section 4.3.3, the on-demand provisioning for each workflow task includes a number of steps: Docker image selection, container creation, dependency provisioning, task downloading and execution, image creation and finally container destruction. All images used to create task containers must be specified in the Service Template by the developer and all required artifacts must be downloaded for each task involved in the workflow deployment.

These provisioning steps are repeated for each task in the workflow (if no task image is created), even if the same task is used more than once or if a dependency package is used repeatedly for similar tasks. Therefore, the deployment of a workflow component

may impose two kinds of delays on the overall performance of the workflow: first, a delay due to the remote downloading and on-line installation and second, the need to repeatedly re-provision a component used in the workflow design.

Moreover, all task images are automatically created during the deployment process whenever new versions of workflows and tasks become available as discussed in section 5.4.4. However, selecting the image for a specific version of workflow or task, to deploying and executing a workflow or an individual task is not a straightforward procedure. As discussed in Chapter 4, the Docker image used to create the container for provisioning a task must be specified by the developer in the Topology Template of the workflow. Further, the images created during the deployment process must be publicly shared by the workflow developer, for example by pushing the images to Docker Hub, yet there is no guarantee that the new tasks/workflows images will be shared and can be reused whenever they are available because they need to be published manually to the public repository by the creator.

Considering the above challenges, we designed new optimization techniques and integrated them into our framework.

6.2.2 Optimization Techniques for Workflow Provisioning

The aim of adding the new optimization to our framework is to automate the sharing and re-usability of workflows and tasks, in addition to supporting the optimization of the workflow provisioning process. It tackles situations when the tasks or other components are repeatedly used by the same workflow or by a different one. In order to realise and explore the optimization, we have implemented these techniques such that they are seamlessly integrated into the existing framework, while the essential components remain unchanged. In other words, the proposed techniques do not require any change to the structure of workflows and tasks. This is because they have been implemented as features that add to the existing core part of the framework (i.e. the lifecycle scripts that manage deployment operations).

One of the core functionalities of our framework is the capability to share workflows/-tasks at various levels, either in the local environment or with public users. As mentioned earlier, Git Hub repositories are used to share source code and other workflow

components with users, and by exploiting Docker users can share deployable components. Furthermore, our framework utilises Git Hub as a version control system to track changes of workflows and tasks so as to increase the degree of workflow reproducibility, where versions for workflows and tasks correspond to the latest tag of the workflow/task repositories in Git Hub (as described in section 5.4.2). In order to maintain workflows and task compatibility with the corresponding images, the name of the image is constructed automatically, comprising a base image name used to provision the container, as well as the workflow/task name and its version. When the image is created, it can be published automatically either in a local repository or Docker Hub for sharing and subsequent use by other users.

However, the need to manually select the correct image to provision a specific version of the task is a challenge since a new image might be created at any time during workflow deployment by other running workflows using the same task. For example, two workflows may have some tasks in common. If they are deployed for the first time nearly simultaneously, each workflow will use a base image to provision the tasks, and in both cases new task images will be automatically created during the deployment process by the AIC. Ideally, if one task was used by a workflow before the other workflow needs it, it would be more efficient if the first task image created by the AIC that could be used by the second workflow.

The first of our optimization extensions introduces a new technique to automate the selection and re-use of a compatible image for task provisioning. The main goal is that tasks are fully provisioned only once and re-used repeatedly. Provisioning a specific version of a task for the first time requires the installation of dependencies and the downloading of the task. During the provisioning process, an image is created by the AIC and should be used for later provisioning of the same version of the task. Therefore, for effective use of a compatible image to provision a task, we have designed and implemented a system that achieves this. It automates the process of publishing the images immediately after creation. In addition, selecting a compatible image is automated by matching the task version with the image name. The same notion can be applied to the deployment of the whole workflow, i.e. a workflow image can be re-used instead of re-provisioning the workflow more than once.

The second optimization focuses on the optimization of the workflow provisioning process by implementing a multi-level cache holding the various workflow components, including deployable entities, task artifacts and dependency packages. We implemented two different types of caching:

1. tasks and workflow images are cached to optimize task and workflow provisioning.
2. essential workflow components such as task artifacts and dependency packages are cached to optimize initial workflow provisioning and image creation, i.e. when the workflow is deployed for the first time and no task or workflow images have been created.

In the following sections we will describe the design and evaluation of these new optimization techniques.

6.3 Transparent Workflow/Task Image Management

One of the main goals of our work is to share effectively not only workflows as a whole but also workflow tasks, so that they are ready-to-use components that can be automatically deployed and effectively used as building blocks in the rapid development of new experiments. The basis for seamless deployment in our framework is effective provisioning, and image management

6.3.1 Just-in-time Task Image Naming, Creation and Selection

To facilitate the process of choosing an appropriate image to provision a workflow task, we automated the process of naming, and the process of selecting compatible images to create the task container. We decided to use a simple yet robust naming convention that combines naming practices enabled by the version control mechanisms of the repositories we use to manage source code and images: GitHub and Docker Hub, respectively.

Workflow tasks in our framework are deployable components, and so the task developer must specify the base image id (image name and version) which they want to use to

execute the task. Usually, it is an id of a pure OS image taken directly from the Docker Hub, but it may also be some specific image that the developer prefers to use instead. The former is much easier to use and it also relieves the task developer from the burden of image maintenance, one of the primary benefits of using our framework.

Given the task's *base image id* we construct the task image id as ***base_image_id.task_name.task_version***. In this way we combine within a single, structured identifier the three elements that impact task provisioning. First, the base image id is important as it will be used to provision the task if no appropriate task image exists. Second, the task name uniquely identifies the repository of a task that is included in the workflow. Third, the task version refers to a specific tag in the task's code repository. The combination of these three elements makes the task image id unique. For example, whenever a new task version is released (i.e. tagged), a new task image will be created, with new id, but workflows that rely on the previous task id will keep using the previous image for that task.

Importantly, when the workflow developer includes a task in their Service Template, they can specify the task's branch name rather than a specific tag. In that case, the framework will automatically detect the latest tag of that branch and use it as the ***task_version*** part of the task image id. That gives workflows the ability to automatically track updates and improvements made by task developers. More details about the benefits that this can bring are discussed in [108].

Our approach to just-in-time image naming and selection is outlined as Algorithm 1. The algorithm iterates over each task in a workflow, identifies the image used to provision the task and makes sure that the image is available in the host execution environment. In line 5 the *task_version* is determined; as mentioned above, it is the appropriate tag from the task repository. Then, following the discussed naming pattern, in line 6 the task image id is set. Based on that identifier, the search process takes place to find the image in the three-level cache (lines 7–14). The search process starts by looking for the task image in the host environment. If that fails, it moves to the second-level cache which is a local repository that can be accessed by authorised users.

Again, if the target image is not found at that cache level, the search will proceed

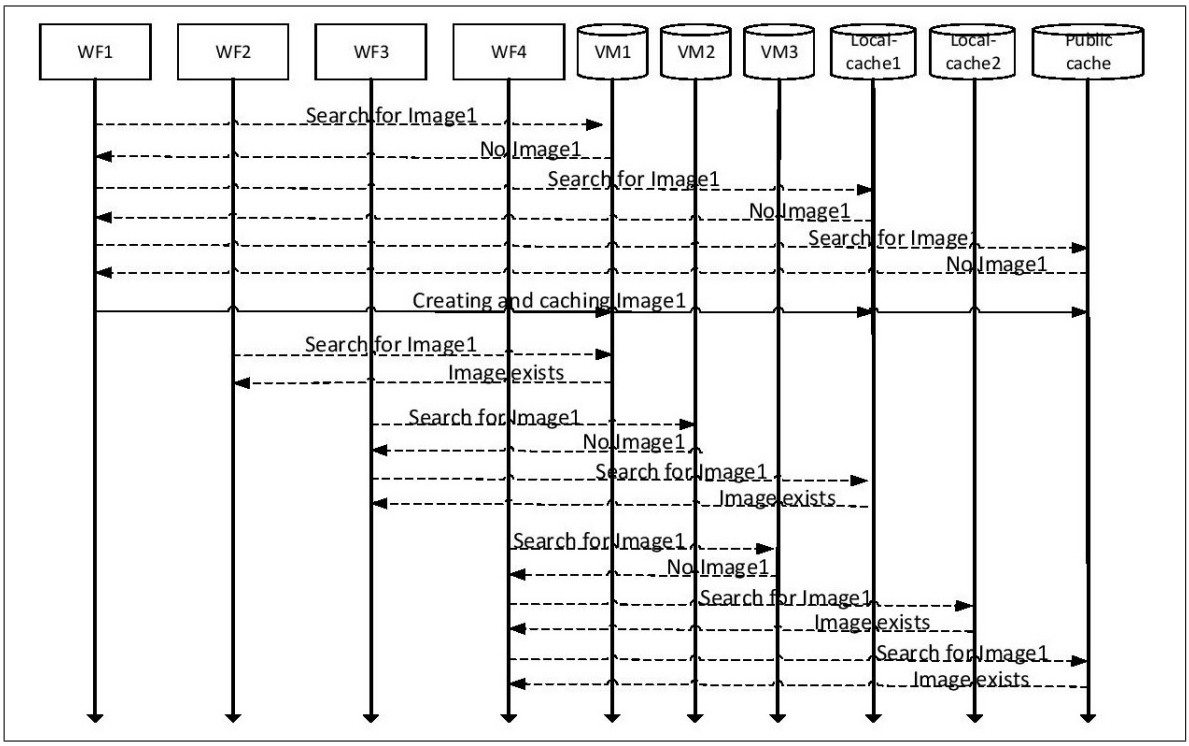

Figure 6.1: Deployment scenario of four workflows with Just-in-time Task Image Selection algorithm.

to the third level – a public repository in Docker Hub. Finally, if no task specific image is found, the task's base image will be used (lines 16–17). The diagram in figure 6.1 depicts how the three-level caching process works during the deployment of four instances of the same workflow (WF1, WF2, WF3 and WF4). Where WF1 and WF2 are running on the same VM (VM1) and shared images through the host environment (On-host1). WF3 is running in VM2 hosted in the same cloud of VM1 therefore it shares images with WF1 and WF2 using a local repository (Local-cache1). Whereas WF4 runs on a different cloud and shares images with the other workflows through the public cache.

Unless the code of a task changes, the same task image is used to execute the task in all workflows in which it is included. This greatly improves the effectiveness of provisioning and performance of workflow enactment, especially if the same task is executed multiple times in a single workflow. Special attention, is however, required in the case when the task image is not yet available, and the task's base image is used in provisioning.

Figure 6.2 depicts the steps our framework follows during workflow task provisioning.

Algorithm 1: Just-in-time Task Image Selection

```
1  t_i – Node Template of i^th workflow task, where i ∈ {1,...,n} and n is the
     number of tasks in workflow W;
2  C_i – container to deploy t_i;
3  I_i – image needed to create C_i;
4  for t_i in W do
5      retrieve task_version of t_i from git;
6      task_image ← base_image_id.task_name.task_version;
7      if task_image available in the host environment then
8          I_i ← task_image;
9      else if task_image available in local cache then
10         copy task_image artifact to the host environment;
11         I_i ← the copied image;
12     else if  task_image available in Docker Hub then
13         docker pull task_image;
14         I_i ← task_image;
15     else
16         docker pull base_image_id;
17         I_i ← base_image_id;
18     end
19 end
```

In the first step Algorithm 1 is run to determine which image is going to be used. If the task image is available, the framework can create a container and immediately proceed to task execution. Otherwise, if the task's base image is used, the framework creates a *base container* and then runs through the Service Template to deploy task dependencies and artifacts. Importantly, just before task execution and for the benefit of any future provisioning requests for that task, an image is created and cached under the unique task image id. The task image is created immediately after downloading task dependencies and artifacts, therefore, no task output data or intermediate data are included, and so the image can safely be used to repeatedly re-execute the task.

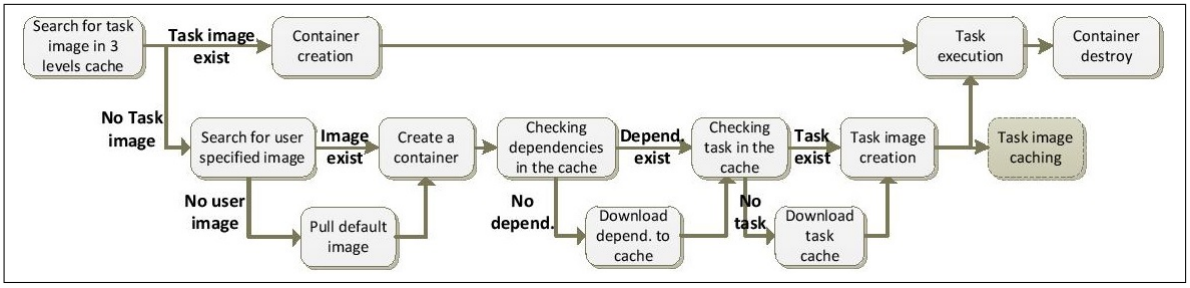

Figure 6.2: Provisioning steps for a workflow task with automatic selection and caching.

Although the process of image creation has some influence on the workflow deployment time, we can save significant amount of time by re-using these images to provision the same task in future executions. In the evaluation section below we present in more detail the benefits and overheads related that effect.

Note that we also use an approach very similar to that presented above for addressing the deployment of single-containers, i.e. when the complete workflow is deployed into one container. Then, instead of looking for a particular image to deploy a task the naming and selection algorithm tries to locate a workflow image following the pattern ***base_image.workflow_name.workflow_version***. The steps to check the availability of the workflow image at the various cache-levels remain the same.

6.3.2 Automatic Image Caching and Sharing

Once the enactment of a workflow ends, all the task images are available in the host execution environment and can be shared with others. Sharing of workflows, and sometimes their components, is supported by a few workflow management system such as Pegasus, Taverna and Galaxy. Yet, we believe that our approach is novel in that it not only allows the structure and description of workflows and tasks to be shared but enables the sharing of ready-to-run components. Users of our workflows and tasks can easily run them in their environment by means of the provided on-click deployment script. Also workflow developers can combine our ready-to-run tasks free from the burden of provisioning the software stack of each task. Additionally, we automate the process of publishing the workflow and task images in order to ensure their immediate availability. As a consequence we can achieve significant reduction in both:

- the effort required by a user to re-execute a workflow,
- the overall workflow deployment and enactment time.

The foundation of image sharing in our framework is the three-level cache, shown in Fig. 6.3, which supports the following use cases. The first level cache in the host environment enables the quickest workflow deployment. If all required images are available in the host, there is no need to download, install and provision any workflows

and workflow tasks. The second level, local repository enables a controlled way of sharing private images within an organisation boundaries but also supports off-line deployment with no access to the Internet/public repository. Lastly, the third level, public repository supports sharing and re-use of images across organisations and also facilitates workflow execution in different clouds platforms.

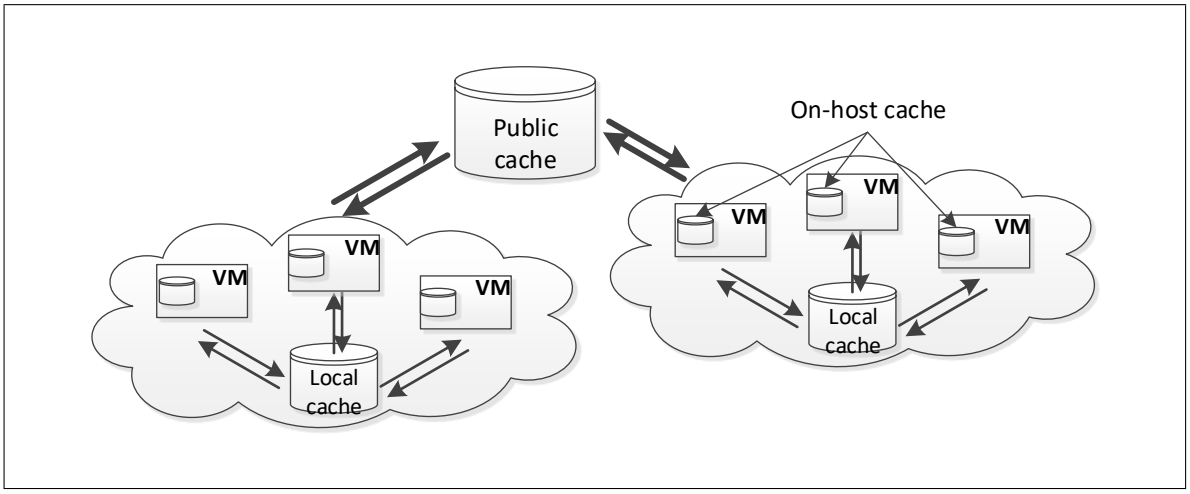

Figure 6.3: Three level caching for task/workflow images.

To implement the cache facility we built on top of the standard tools provided by the Docker platform:

- The first level cache is implemented as the on-host Docker image repository, all images are available in this level immediately when created by AIC.
- The second level cache is an instance of the Docker image repository running on a dedicated machine within an organisation. When the caching option is setted by the user, through input.yaml file, the created images will be transferred to this level.
- The third level, public repository is realised as the Docker Hub public organisation. In parallel with transferring images to the second level, all created images will be pushed to Docker Hub.

In our framework we provide various options for the user to create and share the automatic created images. Using input.yaml file described in section 5.4.1, the user can specify whether to create workflow/task images during the deployment time or

not and the level of sharing either locally or publicly. The input.yaml file contains a parameters for setting these options and used in the lifecycle operations to enable or disable the options.

Unless the user disables this procedure, the images created during workflow deployment are published at the three levels immediately after creation. That ensures the maximum benefits for other workflow executions (and possibly other users) as they may pull ready-to-run images rather than rely on the framework to provision tasks from scratch using the TOSCA descriptors.

6.3.3 Caching Workflow Component Artifacts

The techniques presented above support performance optimization for the provisioning process when tasks and/or workflow images are available. However, when the workflow is deployed for the first time, and to support development of the new tasks, we designed and implemented an additional optimization mechanism. It is a cache of artifacts, such as task code files, dependency tools and libraries; that are essential in building a task and/or workflow image.

A common pattern in scientific workflows is that tasks within one workflow or a family of workflows share common libraries and dependencies. However, following our proposed image naming convention, each of these tasks is packaged in a separate image and follows a separate image creation cycle. This means tasks can not share dependencies.

Therefore, to minimise the time required to create similar images we cache also the workflow component artifacts in the local environment. In this way we can reduce time needed to create the images, which can positively influence workflow enactment. The cache also plays a very useful role for developers as they usually test many more task and workflow versions before they tag an official release to generate a task image.

6.4 Experiments and Evaluation

To evaluate our new optimization techniques, we conducted a set of experiments in which a number of deployment cases have been considered, each with different opti-

mization scenarios.

6.4.1 Experimental Setup

The experiments presented in this section aim to empirically evaluate the effectiveness of our automatic just-in-time selection algorithm, using real-scientific workflows, as well as the influence of the automatic image caching approach on deployment performance, and the impact of caching various workflow components for supporting the performance optimization of the initial deployment process and task image creation.

We ran the experiments on a number of selected real workflows, differing in size, structure and functionality, yet some have tasks and dependencies in common: Neighbour Joining *NJ* (used in Chapter 4 & 5), *WF-1* and *WF-2* with (11, 6, and 8 tasks, respectively). Both *WF-1* and *WF-2* are simple workflows generated to be used in the experiment.

Two environments with a similar configuration have been used to host the experiments:

- Local virtual machine: Ubuntu:14.04 system, 2 CPU Core, 3 GB RAM and 15 GB disk storage
- Google Cloud Platform instances: Debian system, 1 Core CPU, 3.75 GB RAM and 10 GB disk storage

It is important to note that all of the experiments in this chapter are based on the multi-container deployment scenario, i.e. each task is provisioned in a separate container.

6.4.2 The Influence of Task Changes on the Deployment Time

One of the challenges for workflow management systems is the ability to apply changes to part of the workflow without affecting the other unchanged parts. In our work, we have addressed this challenge by (1) isolating the provisioning of each task in a separate Docker container and (2) packaging a full stack of software for provisioning a task in a Docker image.

As described earlier, our deployment framework tracks task versions and uses version numbers as a reference for naming a task image and for selecting the right image. When a new version of a task is detected, the framework will search for the equivalent task image and in the case where no compatible image is available, full provisioning of the task will take place.

To show the validity of our approach for selecting task image and tracking changes, we applied several changes to different tasks of the *NJ* workflow. In this experiment the workflow is executed in different two cases: firstly an image for the changed task has already been created, while in the second case no image exists for it. Therefore, in the first case the task image is used to provision the task and there is no need to install dependencies and download the task; these are however actions that are necessary to execute the task in the second case.

The experiment, as shown in figure 6.4, starts with full deployment **Full Depl** of the *NJ* workflow, i.e. full provisioning of each task and image creation. This is followed by a few executions with task images available locally **Cached-image**. After several executions, a new version of **task1** was created, and no compatible task image was found, which means that the new task version should be fully provisioned. Again, after several executions with **Cashed-image**, a new version of **task1** is discovered, but with a task image available in the public repository. While continuing to re-deploy the workflow, a new version of **task2** is recognised, and a corresponding task image was available and reused in this execution. Finally, a new version of **task3** was found with no related image, so full provisioning of the new task version has been carried out.

The experiment shows that a change in any task does not affect the other parts of the workflow when all tasks were re-provisioned using task images available in the local execution environment (local VM). While there will be an increase in the deployment time when the changed task was re-provisioned either by applying all provisioning steps as depicted in fig 6.2 or a pre-created task image is available in the public repository. Furthermore, the just-in-time selection algorithm has been used in the experiments which show the effect of selecting and re-using the new compatible task image whenever it becomes available. The figure shows that the full deployment of a

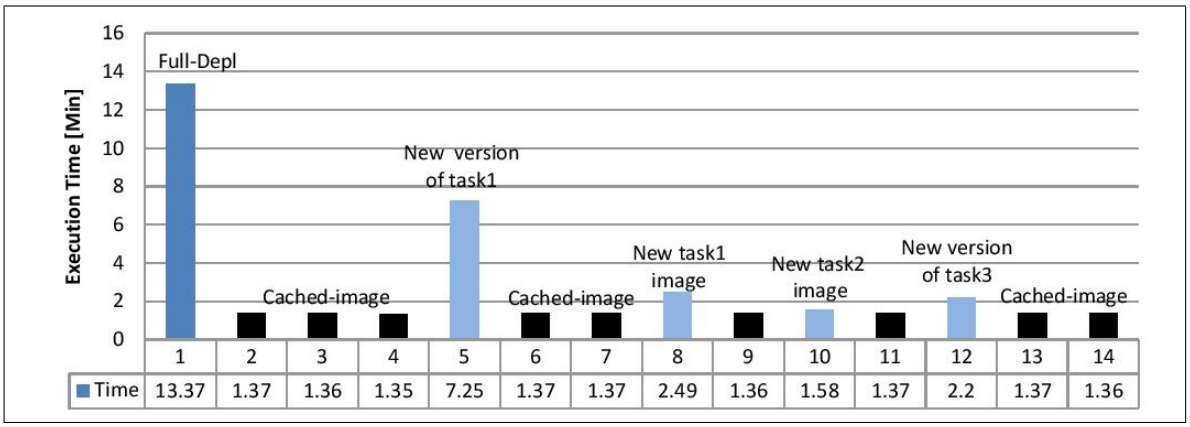

Figure 6.4: Influence of Task Changes on Workflow Execution Time

task takes much longer then when compared with deployment using pre-created task images. This is because of the time required to install task dependencies, especially if they consume a considerable amount of time, such as the first case of provisioning the new version of **task1**.

6.4.3 Task Image Caching for Deployment Optimization

In this experiment, we aim to show the effectiveness of the automatic creation of task images, publishing them in multi-level caches and re-use them. This experiment consists of two parts: the first part shows the influence of creating task images on the overall deployment time. While the second part presents the advantage of multi-level caching and re-using of task images in different cases during the optimization of the workflow provisioning.

Table 6.1: The Influence of Task Image Creation on the Deployment Time.

Deployment Case	Total Deployment Time	Image Creation Time
Cache-off/clear-cache	1883.6	355.5 (18.88%)
Cache-on/clear-cache	967	276.5 (28.6)
No-image-creation	505.2	-

Table 6.1 shows the relative costs for creating task images on the deployment time.

For this, the *NJ* workflow has been executed in three different cases: (1) deployment with caching switched-off and no task image existing in the three-levels of the cache **Cache-off/clear-cache**, (2) deployment with caching switch-on, automatic image

creation and a clear cache, that there was nothing in the cache at the start, **Cache-on/clear-cache** and (3) deployment with caching switched-on but no image creation **No-image-creation**.

The results show that the process of creating task images takes a relatively long time, and so significantly increase the total deployment time. However, this takes place only once, when the workflow is deployed for the first time. In addition, image creation time will be repaid during all subsequent deployments, as those deployment will use the task images instead of re-provisioning the tasks.

The results of the second part of the experiment are shown in figure 6.5 which presents four scenarios for *NJ* deployment. **Cache-on/clear-cache** refers to workflow deployment with a switch-on caching process and empty cache. While **Cache-on/local-cache**, **Cache-on/public-cache** and **Cache-on/full-cache** are deployment scenarios with caching switched-on and task images available in a local repository, public cache, and local execution environment respectively.

As shown in the figure, there is a considerable reduction in the execution time when task images are used instead of re-deploying all tasks. In particular, the optimal minimisation of deployment time can be seen in the case where task images are cached locally in the execution environment. As a result, the overall deployment time is reduced from 1883.6 sec. in the first deployment scenario to 96 sec. for deployment with task images available in the execution environment.

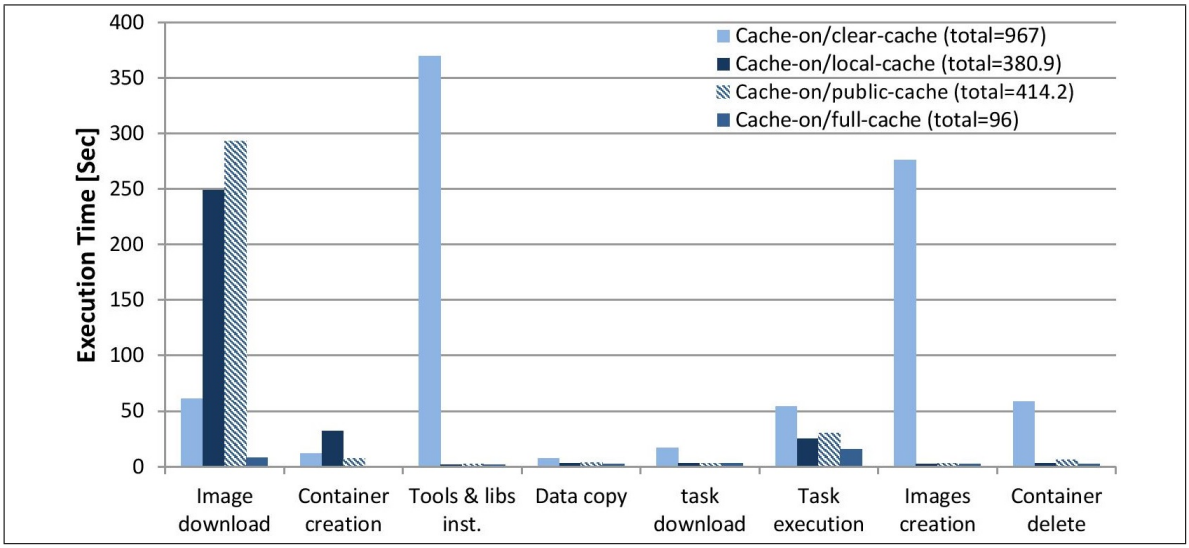

Figure 6.5: Execution Time for *NJ* workflow with Tasks Images Caching.

6.4.4 Sharing Cached Images between Workflows

The experiments presented in the previous section show the effectiveness of the automatic creation, sharing, and re-use of task images for individual deployments of a workflow. In this experiment, we show the impact of using our optimization techniques in the case of concurrent deployment of two instances of the same workflow. The concurrent deployments are run either in the same execution environment, or in different ones. The aim of this part of the experiment is to demonstrate the impact of sharing and re-using task images immediately after creation. Prior to workflow deployment the cache is empty, i.e. no image has been created.

We have conducted this experiment on two scenarios: (1) concurrent executions for two instances of the same workflow in the same VM and (2) concurrent executions of different workflows on different clouds.

6.4.4.1 Concurrent Executions of the Same Workflow

The main reason for automating the process of image sharing and selection is to make sure that an image is created once and can be immediately re-used by the same workflow or by a different one.

To evaluate our approach in this situation, we deployed two instances of the *NJ* workflow concurrently with a time difference between the start of instance executions. The experiments have been conducted with two different execution environments to show how the images can be shared during deployment time and the impact of each individual execution on the other.

In the first case, two instances of *NJ* workflow are executed a number of times on a single machine. The two instances are executed concurrently with time differences between execution initiation. As shown in figure 6.6, the two instances start with no time difference, i.e. the two instances are running in parallel. For the subsequent executions, we triggered the second instance execution after the first execution with increments of one minute as the time interval.

As the two instances are running in the same machine, there is some extra delay in the execution time since they share the same system resources. Apart from the

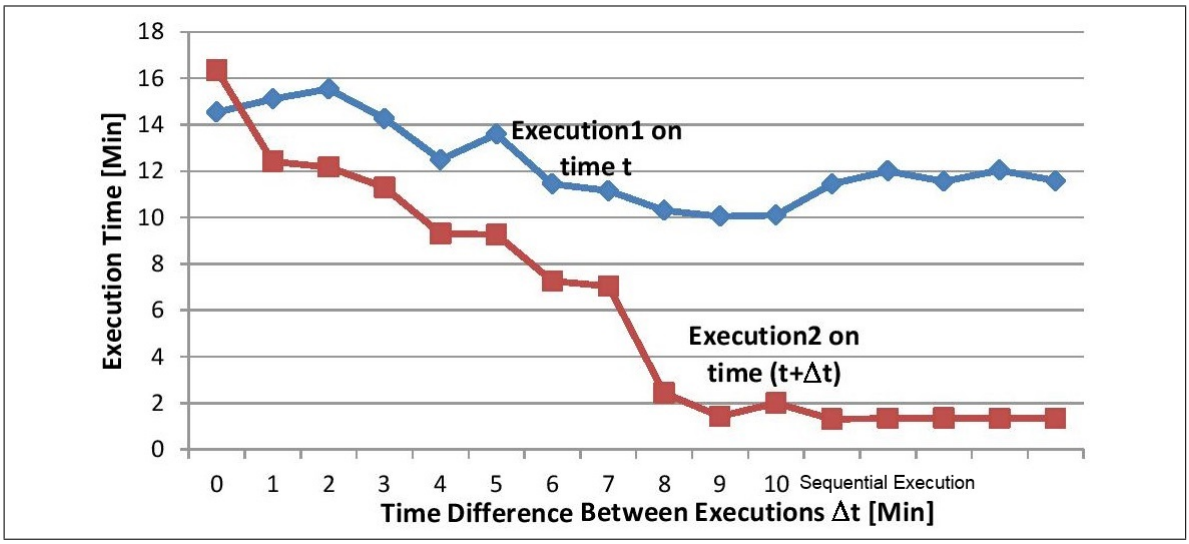

Figure 6.6: The Influence of Sharing Components between Two instances of the NJ Workflow on a Single Machine.

first execution, the figure shows an interesting result for both executions. Specifically, there is a considerable decrease in the execution times, particularly for *Execution2*. The cause of this reduction is the sharing and re-use of workflow components (task artifacts, dependency packages, and task images) between the two instances. The first instance starts provisioning the tasks, which involves downloading and installing these components, and therefore, the second workflow can take advantage of our optimization techniques, which make the components available for utilisation by all subsequent uses. As the difference in the start times for the two instances is increased, greater benefits are gained in the overall execution time. The optimal time is reached when the second execution starts after nine minutes of the first and the execution time is then sequential, i.e. all tasks images have been created by the first workflow execution before the second begins.

Another important result is the reduction of the time for execution of the first instance in some infrequent cases. This happened when *Execution2* reuses some components offered by *Execution1*, and therefore there is no need to download these components and this might speed up its execution because saving the time for downloading the already available components. In addition, when one execution starts creating a task image, the other execution skips the creation process and starts on the next steps. As a consequence, *Execution2* goes ahead of *Execution1* and starts downloading new components, caches them in the environment and offers them to *Execution1*.

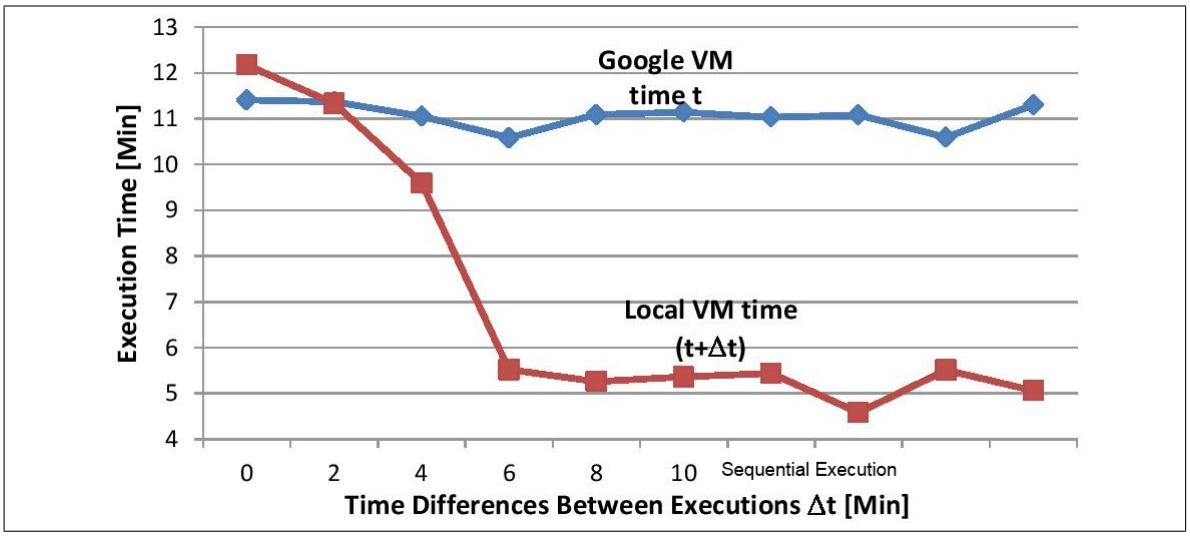

Figure 6.7: The influence of Sharing Tasks Images between Two instances of NJ Workflow on Different Clouds.

In conclusion, these optimization techniques enable sharing of different components between concurrently running workflows on the same environment. This has the effect of reducing the deployment time. Moreover, it can improve the performance of running multiple workflows at the same time on a single machine, for example with multi-core systems.

In the previous experiment, the workflows running on the same machine could share most of the workflow components during the deployment process. In the next experiment we tested another case of running two instances of *NJ* concurrently on different environments (local machine and Google Cloud). Since the workflows are running in different environments, they can only share task images using the public repository (in Docker Hub). This case is useful when two users run concurrently two instances of the same workflow on two different clouds or with a small time difference. Consequently, the tasks images created by one of them can be used by the other, which will save the time of full task provisioning. Whereas with a large time difference between the two instances execution, the workflow instance which starts first will create all task images before the start of the second instance.

The instances were executed several times concurrently with time differences starting from zero and then increasing by interval of one minutes until they ran sequentially. To make the results comparable the cache was emptied prior to each execution.

The results in Figure 6.7 show that the execution times are almost stable for the instance running on Google cloud. This is because the instance is always started first and requires full task provisioning with no task image available. Whereas, the execution times for the second instance running in the local machine decreases gradually as the time difference with the first instance increases. Again, the reason for this is the ability to re-use the task images created by the first instance. All created images are pushed immediately to the public repository to be shared and re-used by others whenever they become available. When the time difference reaches nine minutes, the two instances are executing sequentially (with no overlap in their execution) and the execution times become approximately stable for both instances.

6.4.4.2 Concurrent Executions of Different Workflows

In the previous part of this experiment, we showed the impact of our optimization techniques on the execution time of workflow instances running concurrently. In this part, we conducted a similar experiment, but with different workflows running on different cloud environments. The aim of this experiment is to show how task images can be shared by these workflows that have a number of tasks in common.

We selected three different workflows: *NJ* which has four and five tasks in common with *WF-1* and *WF-2* respectively while *WF-1* and *WF-2* has three tasks in common. In addition, *NJ* and *WF-1* were executed in two different Google VMs while *WF-2* was executed in a local machine at Newcastle University.

The workflows have been executed in three cases differing in their execution order as shown in figures 6.8, figure 6.9, and figure 6.10. In each case, the workflows were executed concurrently with differences in the start time with intervals of one minute, for example in Fig 6.8 *WF-1* starts first then after one minute *WF-2* starts and finally *NJ* starts execution 1 minute after the second workflow. Then the interval increased by 1, therefore, the start time difference becomes two, and so on. Again, it is important to note that the cache was empty before the start of each set of executions.

The result presented in the three figures show stability in the execution time of the first workflow, while there is a gradual decrease in the execution time for the others. This is because the execution of the first workflow included full provisioning for each

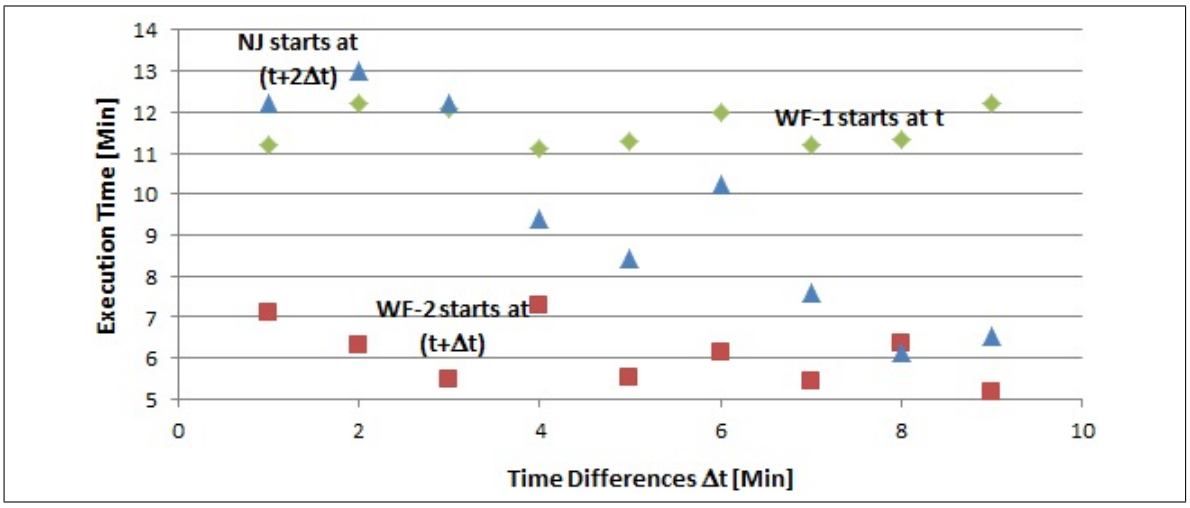

Figure 6.8: Case1: Execution Times for Different Workflows on Different Clouds.

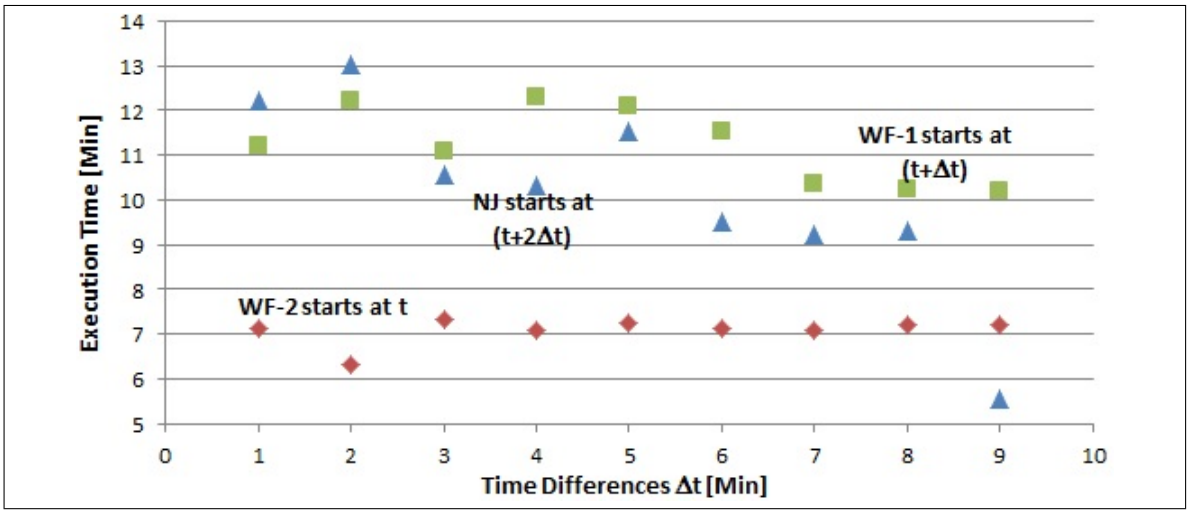

Figure 6.9: Case2: Execution Times for Different Workflows on Different Clouds.

task, creating task images and publishing them to the public repository so that others can re-use them. Furthermore, the reduction in time depends on different factors such as the number of common tasks between the workflows, the time differences between the executions, and the order of the shared tasks in the workflows structure.

From the results shown in Fig 6.8, we can recognise that there is a considerable decrease in the execution time of *NJ* and a slight reduction for *WF-2*. This is because *NJ* is the third running workflow, which means it can reuse the task images created by the first two (up to nine images). While there are fewer tasks shared between *WF-2* and *WF-1*.

In general, the third workflow in each case gains the most benefit from re-using the images, with a reduction in the execution time due to the number of reused images

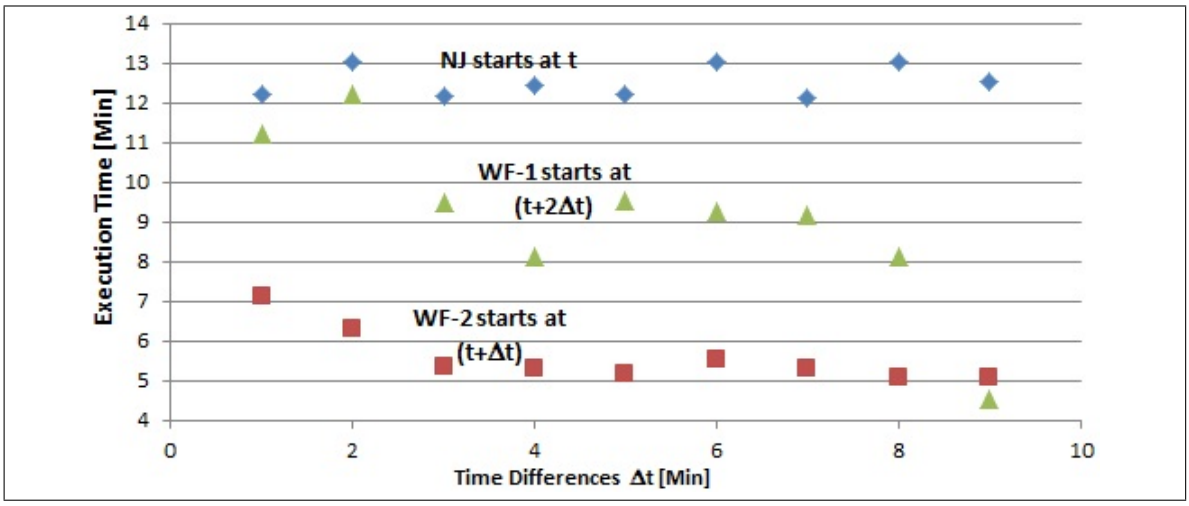

Figure 6.10: Case3: Execution Times for Different Workflows on Different Clouds.

created earlier. The second workflow in the sequence gains some optimization in the deployment-time, depending on which workflow started first.

6.4.5 Optimising Initial Deployment and Image Creation

The previous experiments present the effectiveness of caching task images for optimizing the workflow provisioning and the validity of automatic selection and re-use of these images. However, in the case of initial deployment, when the workflow is deployed for the first time and no task has been created yet, we need to apply our caching technique described in section 6.3.3.

Following this technique, two types of workflow components are cached: task artifacts and dependency packages. Caching these components supports optimization for the initial deployment process and task image creation. In addition, since our framework offers the option to create task images for users, the caching workflow components in this case contribute to the optimization of workflow provisioning.

In this experiment we run the *NJ* workflow in four different cases. The first case is **Cache-off-clear-cache/Image-on**, in which the workflow is deployed with the caching process switched off while creating the task images for the first time. **Cache-on-clear-cache/Image-off** deployment case has caching switched on, with an empty cache and a switched off image creation option, i.e. no task image is created. The other case is **Cache-on-clear-cache/Image-on** deployment, which is the same as the

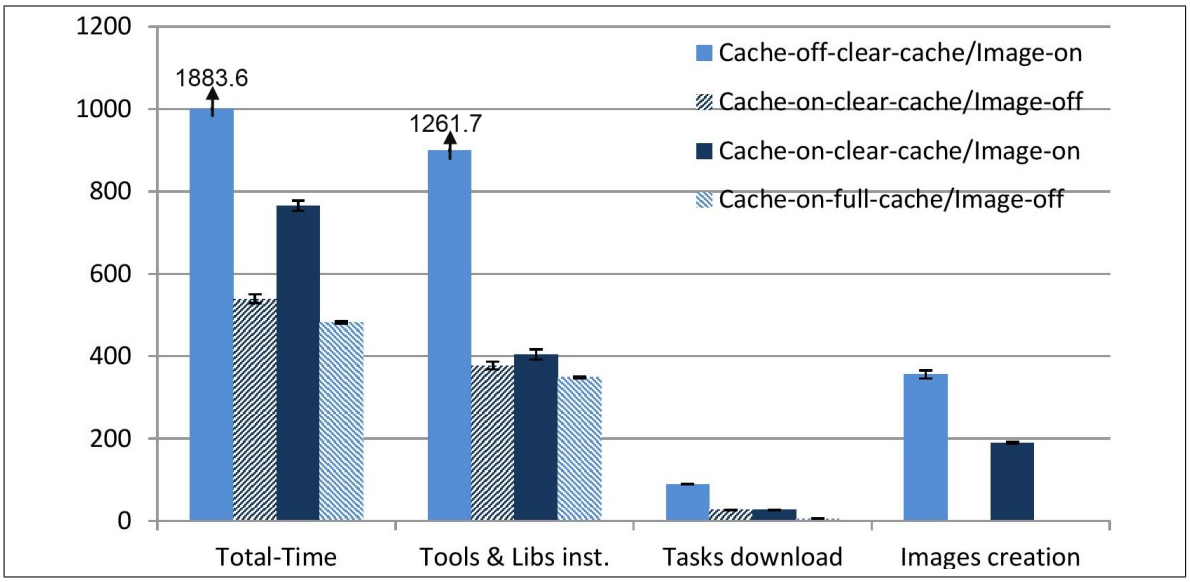

Figure 6.11: Execution Time for NJ Workflow with/without Tasks and Dependencies Caching.

aforementioned case, but with the image creation option switched on. Finally, **Cache-on-full-cache/Image-off** deployment has all tasks and dependencies available in the local cache and has the image creation option switched off.

The workflow was executed ten times in each of the four cases in the local machine and the results show the average of workflow executions in Figure 6.11. It is important to mention that base image is used for creating all containers.

This experiment shows that our approach is able to reduce the overall provisioning time for the workflow in all different cases. This is because all tasks are cached locally so that they are available for all subsequent uses, which eliminates the repeated downloading of the same task. Similarly, all dependency packages required to execute the tasks are cached locally. For example, all tasks in the *NJ* workflow require Java as a dependency to execute the task; therefore, instead of the on-line installation of Java package 11 four times, the package is downloaded once, cached in the local machine and re-used as many times as required. In this way, 10 instances of downloading are saved, whilst also enabling the opportunity for off-line installation.

6.5 Conclusion

The work presented in this chapter further enhances our framework for the reproducibility of scientific workflows. We introduced a set of optimization techniques that adapted our workflow deployment system - a key part of the reproducibility framework - for better performance, in terms of reducing the deployment time and automating the process of sharing, selecting and re-using various workflow components.

We developed a new algorithm for the automatic selection of compatible Docker images for provisioning workflow tasks to facilitate the re-use of these images and support deployment optimization. Furthermore, we have implemented multi-level caching for workflow deployable components, task and workflow images, for optimization purpose and sharing support. In addition, another optimization procedure has been implemented to cache essential workflow components such as task artifacts and dependency packages in the local environments so as to optimize the initial deployment process and image creation.

Our optimization techniques incorporate the tracking of changes in workflow tasks to efficiently select the appropriate image for task provisioning. The new techniques enable the sharing of ready-to-run workflows and tasks. This is not only useful for re-provisioning the workflow but also for concurrent provisioning on the same or different environments. In addition, our experiments showed that the automation of task image selection and sharing speed-up workflow provisioning in a range of different scenarios. These included cases when workflows were executed o the same host, and on different clouds.

Chapter 6: New Techniques for the Optimization of Scientific Workflow Deployment in the Cloud

7

Conclusion

Summary

In this chapter, we summarise the work presented in this thesis, discuss how well the research aims and objectives have been met, and describe open research problems that could motivate future work.

7.1 Thesis Summary

In this thesis, we have presented our new approach for the portable modeling and automatic deployment of scientific workflows in the Cloud. We have also explored how these approaches have been used to develop a new, effective framework to support workflow re-execution and to improve reproducibility. A number of new performance optimization techniques were then designed and evaluated.

Chapter 2 began by giving background information on the main topics related to this thesis, including Cloud Computing, container-based virtualization, scientific workflow and application deployment. Next, details were given on scientific workflows and their management including: modeling, automatic deployment and reproducibility. In addition, the state-of-the-art approaches related to the main focus of this thesis were explored in detail. Many approaches, technologies and tools have been developed that support the modeling and automatic deployment of scientific workflow in the Cloud, as well as improving their reproducibility. We discussed these existing solutions, along with their strengths and weaknesses. However, all the existing mechanisms have limitations and do not fully manage the complexity of deployment and reproducibility of workflows. To address these challenges, we proposed and developed new approaches to model, deploy and effectively support workflow reproducibility.

Chapter 3 described our new approach for modeling scientific workflow using the TOSCA specification. Utilizing TOSCA enables the comprehensive description of scientific workflows, and allows us to capture deployment requirements. We used TOSCA to generate a reusable and portable description of the workflow and its different components, as well as the relationships connecting them. We

showed that TOSCA can be used to model the structure of a workflow in two dimensions spanning both the horizontal links and vertical stack of software components. Further, TOSCA-based modeling formed the basis to achieve automatic deployment for the workflow in the Cloud as described in Chapter 4.

Chapter 4 presented our new approach to the automatic deployment of scientific workflow in the Cloud based on combining TOSCA modeling and container virtualization. The new solution enables building, dynamically deploying and enacting of workflows. We showed that this approach allows workflow deployment to be portable across a range of Cloud and local environments. This allows us to support a number of scenarios for on-demand deployment and the re-usability of different workflow components. In addition, using Docker containers provides isolated execution environments for workflow tasks.

Chapter 5 demonstrated that the reproducibility of scientific workflows is a crucial requirement for scientific experiments, enabling them to be verified, shared and further developed. We presented our framework for improving workflow reusability and reproducibility that leverages our TOSCA-based workflow description, source control and container management, along with the automatic deployment approach presented in Chapter 4. The framework combines both logical and physical preservation techniques: our modeling approach has been used to support logical preservation as a means to describe workflows in a portable way, while container virtualization techniques were used to realise physical preservation, so allowing us to package workflows, tasks and all their dependencies as ready-to-use components. Moreover, our framework used software repositories to manage versioning of workflows and their tasks to facilitate change tracking and to automate the creation of workflow/task images so as to improve the performance of the framework.

Chapter 6 described our work to improve the performance of the reproducibility framework. A number of new optimization techniques have been developed to enhance the performance of our deployment system. We have implemented a new algorithm to automate naming, sharing and selection of workflow/task im-

ages. Further, the new techniques supported automatic caching for workflow components, including the ready-to-use workflow/tasks images, task artifacts and library packages. We proved that using our optimization techniques facilitates tracking changes in workflow tasks and efficiently selecting the correct image for task provisioning. Moreover, the new techniques enabled concurrent deployment and execution of workflows on the same or different environments, which speeds-up workflow provisioning in a number of realistic scenarios.

7.2 Contributions to the Automatic Deployment and Reproducibility of Scientific Workflow

In the introduction to this thesis we stated that the overall aim of this work was "to design, implement and evaluate a system for workflow modeling that offers a portable and re-usable description, so enabling automatic deployment on a range of clouds. The system should efficiently support the re-execution and reproducibility of scientific workflows both in the Cloud and in local computing environments". We now reflect on how we achieved this aim.

In order to enable portable and re-usable definitions of a scientific workflow, and to automate its deployment, a well-defined standard is required. This must fulfil these requirements and enable the capture of the complex deployment and configuration requirements. We have shown that the TOSCA specification can be used to satisfy these requirements by systematically specifying the components and life-cycle management of scientific workflows. This has the advantage of enabling workflow definitions to be portable across clouds, so avoiding the vendor lock-in problem.

Workflows are complex applications consisting of a set of different tasks. Usually, these tasks are heterogeneous components each with their own set of dependencies. This poses challenges for the description and deployment of workflows. Thus, the workflow descriptor needs to include 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 support these requirements, we integrated our TOSCA-based modeling with Docker virtualization to

develop a new approach to automatically deploy the workflow over a range of different Cloud. Using container-based virtualization enables an execution isolation for heterogeneous workflow components and allows the underlying execution environment to be dynamically built and provisioned in accordance with our TOSCA description. We also developed a set of common life-cycle management scripts to manage all required processes to provision workflow tasks together with their dependencies and the hosted execution environments "Docker containers" irrespective of the cloud platform they were running on.

The potential benefits of our TOSCA-based modeling and deployment approach include the portability and re-usability of workflows and their components and ultimately improving workflow reproducibility. Ensuring successful reproducibility of workflows requires both logical and physical preservation. To achieve this, we integrated our workflow description based on TOSCA with source control, container management and the automatic deployment approach. We have designed and developed a new framework to support repeatability and reproducibility of scientific workflow. It combines both logical and physical preservation of the workflow by using the TOSCA-based modeling approach to implement logical preservation and lightweight virtualization to realise physical preservation (packaging workflows, tasks and all their dependencies as Docker images). In addition, the new framework allows workflow and task images to be captured automatically, which improves repeatability, runtime performance, and workflow portability, as well as enabling change tracking.

There is danger that the dynamic provisioning of heterogeneous workflow components may come at the price of additional performance overheads. For this reason, we designed a set of new optimization techniques that facilitate and automate the sharing and re-use of ready-to-run workflows and tasks that have been packaged as Docker images by implementing multi-level caching during the deployment process. Finally, to effectively select a compatible image for deploying a workflow or a task, we implemented a new algorithm that automatically names and selects the images with integration to a version control system.

In summary, to meet the aim of this thesis we explored, integrated and evaluated work in a range of different areas including modelling, virtualization and performance

optimization.

7.3 Future Research Directions

In this section, we described a number of possible directions for future research, which have arisen from the work presented in this PhD research.

7.3.1 Modeling and Invocation of Subworkflows

In Chapter 3, we presented and discussed the capabilities provided by our approach for TOSCA-based modeling of scientific workflow. TOSCA supports definition for the cloud applications and their components using Types, Templates and Topology Template that are re-usable. Moreover, to greatly improve the re-usability, the entire Topology Template may be treated as another Node Type. This can then be utilized to define a workflow that is composed of a number of subworkflows. A separate Topology Template can be defined for each subworkflow and then treated as a Node Type from which a Node Template can be instantiated in the Topology Template of the main workflow. Although the work presented in this thesis can in theory support the modeling of subworkflows, this has not fully explored and the invocation of such a structure has not been implemented.

7.3.2 Supporting the Parallel execution of workflow tasks

Our deployment approach supports the ability to provision each task on an isolated container to be executed separately from the other tasks. However, our approach does not offer parallel provisioning for unrelated tasks which have no dependencies - either direct or indirect - between them. The reason behind this limitation is that our deployment approach uses Cloudify as a run-time environment for TOSCA-based workflow modelling, and it does not offer parallel execution. Supporting such a provisioning scenario would enhance the performance of workflow execution.

7.3.3 Modeling Various Types of Scientific Workflow

We evaluated the modelling and deployment approaches that are essential parts of our reproducibility framework using a number of existing workflows previously designed in e-Science Central. Future work might consider investigating to what extent our approaches can be used to model and deploy legacy workflows designed in other scientific workflow management systems like Pegasus and Taverna. This would allow us to understand the generality of our approach. Differences between the workflow management systems that might require further investigation including service-based management, data-intensive processing and different execution environment targets, such as a remote physical cluster or grid.

7.3.4 Capturing Provenance Data for Comprehensive Reproducibility

Data provenance in workflows is captured as a set of dependencies between data elements [88]. It may be used for interpreting data and providing reproducible results, and also for troubleshooting and optimizing workflow efficiency. Our TOSCA-based workflow descriptions may be considered as a detailed prospective provenance document, and therefore one possibility of future research direction is to extend the reproducibility framework presented in this thesis to improve the degree of reproducibility of scientific workflows by collecting and exploiting provenance data. An extension could be designed and added to capture retrospective provenance information for workflows and tasks, including detailed information such as the execution environment and intermediate results, and this can then be combined with their development history, building on the work presented in this thesis.

7.3.5 Distributed Workflow Enactment on Hybrid Cloud

Using our deployment approach, whole workflows can be deployed on different cloud and local environments, i.e. all tasks can be deployed in the same cloud or in a local VM. This approach can be extended to support large-scale, distributed workflows that span clouds (i.e. tasks spread over both public and private clouds: Azure, Google,

AWS etc.). Next an extension could be designed for automatically deploying the workflow over those clouds and executing it. Criteria used in selecting the deployment plan could include fulfilling the security requirements for processing the data by placing some privacy-sensitive tasks and data on private clouds [137] and also co-locating tasks on clouds that host the data they need to consume.

7.3.6 Fault-Tolerance and recovery Strategies

Only small number of the existing systems for scientific workflow management provide explicit mechanisms to handle different types of hardware and/or software failures [123]. However, fault tolerance is important when mission-critical workflows are deployed on distributed environments. To address those issues a fault tolerance and recovery techniques might be embodied in the deployment system to reduce the impact of faults. For example, this might be achieved by detecting and re-provisioning failed tasks.

8

Appendix

8.1 Appendix A

Topology Template of a workflow

Listing 8.1: Topology Template for a sample workflow.

```
tosca_definitions_version: cloudify_dsl_1_0

imports:
  - http://www.getcloudify.org/spec/cloudify/3.1/types.yaml
  - https://raw.githubusercontent.com/rawaqasha/e-sc-cloudify/master/
     esc_nodetypes.yaml

inputs:

  input-dir:
      description: >
        The dir path of the input files
      default: '~/input'

  input-file:
      description: >
        input file for importFile1
      default: file.jpg

  docker-image:
      description: >
        Docker image to be used for container building
      default: 'ubuntu:14.04'

  create_image:
      description: >
        an option to create Docker images
      default: 'False'

node_types:

  docker_container:
    derived_from: cloudify.nodes.Root
    properties:
      image_name:
        type: string
        default: { get_input: docker-image }
      container_ID:
        type: string
        default: container1

node_templates:

  host:
    type: cloudify.nodes.Compute
    properties:
      ip: localhost
      install_agent: false

  starterBlock:
```

```
  type: cloudify.nodes.ApplicationModule
  interfaces:
    cloudify.interfaces.lifecycle:
      create:
        implementation: Core-LifecycleScripts/start-inhost.sh
        inputs:
          process:
            args: [FileZip]
  relationships:
    - type: cloudify.relationships.contained_in
      target: host

container1:
  type: docker_container
  properties:
    container_ID: container1
  interfaces:
    cloudify.interfaces.lifecycle:
      create:
        implementation: Core-LifecycleScripts/container.sh
        inputs:
          process:
            args: [FileZip]
  relationships:
    - type: cloudify.relationships.contained_in
      target: host
    - type: cloudify.relationships.depends_on
      target: starterBlock

Java:
  type: spec_library
  properties:
    lib_name: default-jdk
  interfaces:
    cloudify.interfaces.lifecycle:
      create:
        implementation: scripts/java-install2.sh
        inputs:
          process:
            args: [container1, FileZip]
  relationships:
    - type: cloudify.relationships.depends_on
      target: container1
    - type: cloudify.relationships.contained_in
      target: container1

ImportFile:
  type: importfile
  properties:
    block_description: import file
    block_name: importfile1.jar
    block_category: File Management
    service_type: block
    Source: { get_input: input-file }
  interfaces:
    cloudify.interfaces.lifecycle:
```

```
      create:
        implementation: Core-LifecycleScripts/task-download-multi.
            sh
        inputs:
          process:
           args: [{ get_input: create_image }, container1, 'https
              ://github.com/rawaqasha/eScBlocks-host/raw/master/
              importfile1.jar']
      configure:
        implementation: scripts/get-input.sh
        inputs:
          process:
           args: [FileZip, container1, { get_input: input-dir }, {
              get_input: input-file }]
      start:
        implementation: Core-LifecycleScripts/task-deploy.sh
        inputs:
          process:
           args: [FileZip, container1, { get_input: input-file }]
  relationships:
    - type: cloudify.relationships.contained_in
      target: container1
    - type: cloudify.relationships.depends_on
      target: Java

container2:
  type: docker_container
  properties:
    container_ID: container2
  interfaces:
    cloudify.interfaces.lifecycle:
      start:
        implementation: Core-LifecycleScripts/container.sh
        inputs:
          process:
            args: [FileZip]
  relationships:
    - type: cloudify.relationships.contained_in
      target: host
    - type: cloudify.relationships.depends_on
      target: ImportFile

Java2:
  type: spec_library
  properties:
    lib_name: default-jdk
  interfaces:
    cloudify.interfaces.lifecycle:
      create:
        implementation: scripts/java-install2.sh
        inputs:
          process:
            args: [container2, FileZip]
  relationships:
    - type: cloudify.relationships.contained_in
      target: container2
```

```
ZipFile:
  type: zipFile
  properties:
    block_description: zip the input file
    block_name: filezip2.jar
    block_category: File Management
    service_type: block
  interfaces:
    cloudify.interfaces.lifecycle:
      create:
        implementation: Core-LifecycleScripts/task-download-multi.
            sh
        inputs:
          process:
           args: [{ get_input: create_image }, container2, 'https
              ://github.com/rawaqasha/eScBlocks-host/raw/master/
              filezip2.jar']
      configure:
        implementation: Core-LifecycleScripts/containers-clean.sh
        inputs:
          process:
           args: [container1]
      start:
        implementation: Core-LifecycleScripts/task-deploy.sh
        inputs:
          process:
           args: [FileZip, container2, rawa1975]
  relationships:
    - type: cloudify.relationships.contained_in
      target: container2
    - type: cloudify.relationships.depends_on
      target: Java2
    - type: block_link
      target: ImportFile
      source_interfaces:
        cloudify.interfaces.relationship_lifecycle:
          preconfigure:
            implementation: Core-LifecycleScripts/datacopy.sh
            inputs:
              process:
                args: [ImportFile/output-1, ZipFile/input-1,
                    FileZip, container2]

container3:
  type: docker_container
  properties:
    container_ID: container3
  interfaces:
    cloudify.interfaces.lifecycle:
      start:
        implementation: Core-LifecycleScripts/container.sh
        inputs:
          process:
            args: [FileZip]
  relationships:
```

```
    - type: cloudify.relationships.contained_in
      target: host
    - type: cloudify.relationships.depends_on
      target: ZipFile

Java3:
  type: spec_library
  properties:
    lib_name: default-jdk
  interfaces:
    cloudify.interfaces.lifecycle:
      create:
        implementation: scripts/java-install2.sh
        inputs:
          process:
            args: [container3, FileZip]
  relationships:
    - type: cloudify.relationships.contained_in
      target: container3

ExportFiles:
  type: exportfiles
  properties:
    block_description: export files
    block_name: exportfiles1.jar
    block_category: File Management
    service_type: block
  relationships:
    - type: cloudify.relationships.contained_in
      target: container3
    - type: block_link
      target: ZipFile
      source_interfaces:
        cloudify.interfaces.relationship_lifecycle:
          preconfigure:
            implementation: Core-LifecycleScripts/datacopy.sh
            inputs:
              process:
                args: [ZipFile/output-1, ExportFiles/file-list,
                   FileZip, container3]
    - type: cloudify.relationships.depends_on
      target: Java3
  interfaces:
    cloudify.interfaces.lifecycle:
      create:
        implementation: Core-LifecycleScripts/task-download-multi.
           sh
        inputs:
          process:
           args: [{ get_input: create_image }, container3, 'https
              ://github.com/rawaqasha/eScBlocks-host/raw/master/
              exportfiles1.jar']
      configure:
        implementation: Core-LifecycleScripts/containers-clean.sh
        inputs:
          process:
```

```
            args: [container2]
        start:
          implementation: Core-LifecycleScripts/task-deploy.sh
          inputs:
            process:
             args: [FileZip, container3]

finalBlock:
  type: cloudify.nodes.ApplicationModule
  interfaces:
    cloudify.interfaces.lifecycle:
      configure:
        implementation: Core-LifecycleScripts/containers-clean.sh
        inputs:
          process:
            args: [container3]
      create:
        implementation: Core-LifecycleScripts/final-inhost.sh
        inputs:
          process:
            args: [FileZip]
  relationships:
    - type: cloudify.relationships.contained_in
      target: host
    - type: cloudify.relationships.depends_on
      target: ExportFiles
```

Bibliography

[1] *Amazon Elastic Compute Cloud (Amazon EC2).* Amazon Inc., http://aws.amazon.com/ec2.

[2] *Google App Engine.* Google Inc., http://code.google.com/appengine.

[3] *Salesforce.com.* Salesforce Inc, salesforce.com.

[4] E Afgan, D Baker, N Coraor, B Chapman, A Nekrutenko, and J Taylor. Galaxy cloudman: delivering cloud compute clusters. *BMC Bioinformatics*, 11(12):S4, 2010.

[5] E Afgan, D Baker, M van den Beek, and et al. The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2016 update. *Nucleic Acids Research*, 44(May), 2016.

[6] 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*, 27(16):4330–4343, 2015.

[7] P Amstutz, M. R Crusoe, N Tijanić, B Chapman, J Chilton, M Heuer, A Kartashov, D Leehr, H Ménager, M Nedeljkovich, M Scales, S Soiland-Reyes, and L Stojanovic. Common workflow language, v1.0. 2017.

[8] S Arabas, M. R Bareford, L. R de Silva, I. P Gent, B. M Gorman, M Hajiarabderkani, T Henderson, L Hutton, A Konovalov, L Kotthoff, C McCreesh, M Nacenta, R. R Paul, K. E. J Petrie, A Razaq, D Reijsbergen, and K Takeda. Case Studies and Challenges in Reproducibility in the Computational Sciences. *CoRR*, abs/1408.2:1–14, Aug 2014.

[9] M Armbrust, A Fox, R Griffith, A. D Joseph, R. H Katz, A Konwinski, G Lee, D. A Patterson, A Rabkin, I Stoica, *et al.* Above the clouds: A berkeley view of cloud computing. Technical report, UCB/EECS-2009-28, EECS Department, University of California, Berkeley, 2009.

[10] M Armbrust, I Stoica, M Zaharia, A Fox, R Griffith, A. D Joseph, R Katz, A Konwinski, G Lee, D Patterson, and A Rabkin. A view of cloud computing. *Communications of the ACM*, 53(4):50, 2010.

[11] M Atay, A Chebotko, D Liu, S Lu, and F Fotouhi. Efficient schema-based xml-to-relational data mapping. *Inf. Syst.*, 32(3):458–476, May 2007.

[12] B Balis. Hyperflow: A model of computation, programming approach and enactment engine for complex distributed workflows. *Future Generation Computer Systems*, 55:147 – 162, 2016.

[13] B Balis, K Figiela, Malawski, and et al. A Lightweight Approach for Deployment of Scientific Workflows in Cloud Infrastructures. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9573, pages 281–290. 2016.

[14] A Banati, P Kacsuk, and M Kozlovszky. Four level provenance support to achieve portable reproducibility of scientific workflows. In *38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, number May, pages 241–244. IEEE, May 2015.

[15] P Barham, B Dragovic, K Fraser, S Hand, T Harris, A Ho, R Neugebauer, I Pratt, and A Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[16] A Barker and J van Hemert. *Scientific Workflow: A Survey and Research Directions*, pages 746–753. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[17] K Baxley and J. D la Rosa. Deploying Workloads With Juju and MAAS, Dell Inc., Technical White Paper, 2013.

[18] K Belhajjame, J Zhao, D Garijo, M Gamble, K Hettne, R Palma, E Mina, O Corcho, J. M Gómez-Pérez, S Bechhofer, G Klyne, and C Goble. Using a suite of ontologies for preserving workflow-centric research objects. *Web Semantics: Science, Services and Agents on the World Wide Web*, 32:16–42, 2015.

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

[20] T Binz, U Breitenbücher, O Kopp, and F Leymann. Tosca: Portable automated deployment and management of cloud applications. In *Advanced Web Services*, pages 527–549. Springer New York, New York, NY, 2014.

[21] C Boettiger. An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1):71–79, January 2015.

[22] P Bonnet, S Manegold, M Bjørling, W Cao, J Gonzalez, J Granados, N Hall, S Idreos, M Ivanova, R Johnson, D Koop, T Kraska, R Müller, D Olteanu, P Papotti, C Reilly, D Tsirogiannis, C Yu, J Freire, and D Shasha. Repeatability and workability evaluation of sigmod 2011. *SIGMOD Rec.*, 40(2):45–48, September 2011.

[23] U Breitenbücher, T Binz, O Kopp, F Leymann, and M Wieland. *Context-Aware Provisioning and Management of Cloud Applications, Cloud Computing and Services Sciences*, pages 151–168. Springer International Publishing, 2015.

[24] A Brogi and J Soldani. *Matching Cloud Services with TOSCA, Advances in Service-Oriented and Cloud Computing*, pages 218–232. Springer Berlin Heidelberg, 2013.

[25] O Bushehrian and R. G Baghnavi. Deployment optimization of software objects by design-level delay estimation. *Journal of Supercomputing*, 65(3):1243–1263, 2013.

[26] R Buyya and S Venugopal. The gridbus toolkit for service oriented grid and utility computing: an overview and status report. In *1st IEEE International Workshop on Grid Economics and Business Models, GECON 2004*, pages 19–66, April 2004.

[27] M Caballer, D Segrelles, G Moltó, and I Blanquer. A platform to deploy customized scientific virtual infrastructures on the cloud. *Concurrency and Computation: Practice and Experience*, 27(16):4318–4329, 2015.

[28] M Cafaro and G Aloisio. *Grids, Clouds and Virtualization*. Computer Communications and Networks. Springer London, London, 2011.

[29] 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*, 65(Supplement C):153 – 168, 2016. Special Issue on Big Data in the Cloud.

[30] J Cała, H Hiden, S Woodman, and P Watson. Cloud computing for fast prediction of chemical activity. *Future Generation Computer Systems*, 29(7):1860–1869, sep 2013.

[31] Cardiff University. *Triana*. http://www.trianacode.org/.

[32] J. V Carri, C. D Alfonso, and M Caballer. A Generic Catalog and Repository Service for Virtual Machine Images. In *2nd International ICST Conference on Cloud Computing (CloudComp 2010)*, number Vm, 2010.

[33] E Casalicchio and V Perciballi. Measuring docker performance: What a mess!!! In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ICPE '17 Companion, pages 11–16, New York, NY, USA, 2017. ACM.

[34] R Chamberlain and J Schommer. Using docker to support reproducible research. *figshare*, 1101910, 2014.

[35] F Chirigati, D Shasha, and J Freire. ReproZip : Using Provenance to Support Computational Reproducibility. *USENIX Workshop on the Theory and Practice of Provenance*, 2013.

[36] S Cohen-Boulakia, K Belhajjame, O Collin, J Chopard, C Froidevaux, A Gaignard, K Hinsen, P Larmande, Y. L Bras, F Lemoine, F Mareuil, H Ménager, C Pradal, and C Blanchet. Scientific workflows for computational reproducibility in the life sciences: Status, challenges and opportunities. *Future Generation Computer Systems*, 75(Supplement C):284 – 298, 2017.

[37] P Czarnul. Modeling, run-time optimization and execution of distributed workflow applications in the JEE-based BeesyCluster environment. *Journal of Supercomputing*, 63(1):46–71, 2013.

[38] A Dearle. Software Deployment, Past, Present and Future. In *Future of Software Engineering (FOSE '07)*, pages 269–284. IEEE, May 2007.

[39] E Deelman and Y Gil. Workshop on the challenges of scientific workflows. *Information Sciences Institute*, 2006.

[40] E Deelman, D Gannon, M Shields, and I Taylor. Workflows and e-Science: An overview of workflow system features and capabilities. *Future Generation Computer Systems*, 25(5):528–540, May 2009.

[41] E Deelman, G Singh, H Su, J Blythe, Y Gil, C Kesselman, G Mehta, K Vahi, and G. B Berriman. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13:219–237, 2005.

[42] 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*, 46(Supplement C):17 – 35, 2015.

[43] E Deelman, K Vahi, M Rynge, G Juve, R Mayani, and R. F da Silva. Pegasus in the Cloud: Science Automation through Workflow Technologies, IEEE Internet Computing. 20(1):70–76, 2016.

[44] DMTF. Distributed Management Task Force , COMMON INFORMATION MODEL (CIM). 2005.

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

[46] J Fink. Docker : a Software as a Service , Operating System-Level Virtualization Framework. *Code4Lib*, (25):3–5, 2014.

[47] I Foster, Y Zhao, I Raicu, and S Lu. Cloud computing and grid computing 360-degree compared. In *2008 Grid Computing Environments Workshop*, pages 1–10, Nov 2008.

[48] J Freire, P Bonnet, and D Shasha. Computational reproducibility: state-of-the-art, challenges, and database research opportunities. *Proceedings of the 2012 ACM SIGMOD*, pages 593–596, 2012.

[49] W Gao, H Jin, S Wu, X Shi, and J Yuan. Effectively deploying services on virtualization infrastructure. *Frontiers of Computer Science*, pages 398–408, July 2012.

[50] M Gavish and D Donoho. A universal identifier for computational results. *Procedia Computer Science*, 4:637 – 647, 2011.

[51] D Georgakopoulos, M Hornick, and A Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, Apr 1995.

[52] 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*, pages 25–32. IEEE, nov 2014.

[53] C 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*, 38(May):677–682, 2010.

[54] 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*, 11(8):R86, January 2010.

[55] K Hasham, K Munir, R McClatchey, and J Shamdasani. *Re-provisioning of Cloud-Based Execution Infrastructure Using the Cloud-Aware Provenance to Facilitate Scientific Workflow Execution Reproducibility*, pages 74–94. Springer International Publishing, Cham, 2016.

[56] S He, L Guo, Y Guo, C Wu, M Ghanem, and R Han. Elastic Application Container: A Lightweight Approach for Cloud Resource Provisioning. *2012 IEEE 26th International Conference on Advanced Information Networking and Applications*, pages 15–22, mar 2012.

[57] K Hettne, K Wolstencroft, K Belhajjame, C Goble, E Mina, H Dharuri, L Verdes-Montenegro, J Garrido, D De Roure, and M Roos. Best practices for workflow design: How to prevent workflow decay. *CEUR Workshop Proceedings*, 952, 2012.

[58] H Hiden, S Woodman, P Watson, and J Cala. Developing cloud applications using the e-Science Central platform. *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences*, 371(1983):1–12, January 2013.

[59] C Hoffa, G Mehta, T Freeman, E Deelman, K Keahey, B Berriman, and J Good. On the Use of Cloud Computing for Scientific Workflows. *2008 IEEE Fourth International Conference on eScience*, pages 640–645, dec 2008.

[60] B Howe. Virtual Appliances, Cloud Computing, and Reproducible Research. *Computing in Science & Engineering*, 14(4):36–41, July 2012.

[61] HTCondor. DAGMan: A Directed Acyclic Graph Manager, 2005.

[62] F Jiang, C Castillo, C Schmitt, A Mandal, P Ruth, and I Baldin. Enabling workflow repeatability with virtualization support. *Proceedings of the 10th Workshop on Workflows in Support of Large-Scale Science - WORKS '15*, pages 1–10, 2015.

[63] G Juve, A Chervenak, E Deelman, S Bharathi, G Mehta, and K Vahi. Characterizing and profiling scientific workflows. *Future Generation Computer Systems*, 29(3):682–692, mar 2013.

[64] G Juve and E Deelman. Scientific workflows and clouds. *Crossroads*, 16(3):14–18, March 2010.

[65] G Juve and E Deelman. Automating Application Deployment in Infrastructure Clouds. *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, pages 658–665, November 2011.

[66] G Juve, E Deelman, K Vahi, G Mehta, B Berriman, B. P Berman, and P Maechling. Scientific workflow applications on Amazon EC2. *2009 5th IEEE International Conference on E-Science Workshops*, pages 59–66, December 2009.

[67] G Katsaros, M Menzel, A Lenk, J. R Revelant, R Skipp, and J Eberhardt. Cloud Application Portability with TOSCA, Chef and Openstack. In *2014 IEEE International Conference on Cloud Engineering*, pages 295–302. IEEE, March 2014.

[68] G Kecskemeti, P Kacsuk, G Terstyanszky, T Kiss, and T Delaitre. Automatic Service Deployment Using Virtualisation. In *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 628–635. IEEE, February 2008.

[69] G Kecskemeti, G Terstyanszky, P Kacsuk, and Z Neméth. An approach for virtual appliance distribution for service deployment. *Future Generation Computer Systems*, 27(3):280–289, March 2011.

[70] H Khazaei, C Barna, N Beigi-mohammadi, and M Litoiu. Efficiency Analysis of Provisioning Microservices. In *2016 IEEE 8th International Conference on Cloud Computing Technology and Science, 2016 IEEE (CloudCom'16)*, pages 261–268, 2016.

[71] S Kolb, J org Lenhard, and G Wirtz. Application Migration Effort in the Cloud-the case of cloud platforms. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 41–48, 2015.

[72] M Kostoska, I Chorbev, and M Gusev. Creating portable TOSCA archive for iKnow University Management System. In *Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 761–768, September 2014.

[73] G Kougka, A Gounaris, and A Simitsis. The many faces of data-centric workflow optimization: A survey. *arXiv preprint arXiv:1701.07723*, 2017.

[74] Z Kozhirbayev and R. O Sinnott. A performance comparison of container-based technologies for the cloud. *Future Generation Computer Systems*, 68:175 – 182, 2017.

[75] 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*, pages 61–68. IEEE, December 2013.

[76] H Li. *Introducing Windows Azure*. Apress, Berkely, CA, USA, 2009.

[77] B Liu, J Li, and C Liu. Cloud-based bioinformatics workflow platform for large-scale next-generation sequencing analyses. *Journal of biomedical informatics*, January 2014.

[78] 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*, pages 1087–1095, November 2012.

[79] K Liu, K Aida, S Yokoyama, and Y Masatani. Flexible container-based computing platform on cloud for scientific workflows. In *2016 International Conference on Cloud Computing Research and Innovations (ICCCRI)*, pages 56–63, May 2016.

[80] X Liu, D Yuan, G Zhang, W Li, D Cao, Q He, J Chen, and Y Yang. *The Design of Cloud Workflow Systems*. Springer Briefs in Computer Science. Springer New York, New York, NY, 2012.

[81] Y Liu, I Gorton, A Wynne, and A Kullkarni. Scientific workflows composition and deployment on SOA frameworks. In *Proceedings of 2011 IEEE 6th International Symposium on Service Oriented System (SOSE)*, pages 134–139. IEEE, dec 2011.

[82] B Ludäscher, I Altintas, C Berkley, and et al. Scientific workflow management and the kepler system: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(10):1039–1065, August 2006.

[83] M Malawski, G Juve, E Deelman, and J Nabrzyski. Cost- and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11, Nov 2012.

[84] M Malawski, G Juve, E Deelman, and J Nabrzyski. Algorithms for cost- and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds. *Future Generation Computer Systems*, 48(Supplement C):1 – 18, 2015. Special Section: Business and Industry Specific Cloud.

[85] P Mates, E Santos, J Freire, and C Silva. Crowdlabs: Social analysis and visualization for the sciences. In *Scientific and Statistical Database Management*, volume 6809 LNCS, pages 555–564. Springer Berlin Heidelberg, 2011.

[86] W. I Matters. Reproducible Research. Addressing the need for data and code sharing in computational science. *Computing in Science & Engineering*, pages 8–13, 2010.

[87] R Mayer and A Rauber. A Quantitative Study on the Re-executability of Publicly Shared Scientific Workflows. In *2015 IEEE 11th International Conference on e-Science*, pages 312–321. IEEE, aug 2015.

[88] D McGuinness, J Michaelis, L Moreau, O Hartig, and J Zhao. Provenance and Annotation of Data and Processes. *Ipaw*, 5272:78–90–90, 2008.

[89] P Mell and T Grance. The NIST definition of cloud computing. *NIST Special Publication*, 145:7, 2011.

[90] H Meng, R Kommineni, Q Pham, R Gardner, T Malik, and D Thain. An invariant framework for conducting reproducible computational science. *Journal of Computational Science*, 9:137–142, 2015.

[91] D Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[92] J. M Miller and B A. *Solution Deployment Descriptor (SDD), Part 1 : An emerging standard for deployment artifacts*. OASIS, 2008.

[93] D Milojičić, I. M Llorente, and R. S Montero. Opennebula: A cloud management tool. *IEEE Internet Computing*, 15(2):11–14, March 2011.

[94] P Missier, S Woodman, H Hiden, and P Watson. Provenance and data differencing for workflow reproducibility analysis. *Concurrency Computation Practice and Experience*, (October), 2013.

[95] A. H Neubauer and Bertram. Deployment and Configuration of Distributed systems. In *Proceedings of the 4th International SDL and MSC Conference on System Analysis and Modeling*, pages 1–16, Berlin, Heidelberg, 2005. Springer.

[96] F Neubauer, A Hoheisel, and J Geiler. Workflow-based grid applications. *Future Gener. Comput. Syst.*, 22(1-2):6–15, January 2006.

[97] D Nurmi, R Wolski, C Grzegorczyk, G Obertelli, S Soman, L Youseff, and D Zagorodnov. Eucalyptus : A Technical Report on an Elastic Utility Computing Archietcture Linking Your Programs to Useful Systems UCSB Computer Science Technical Report Number 2008-10. *Control*, 10:2008, 2008.

[98] OASIS. sdd-v2.0-csd01, 2010.

[99] OASIS. Topology and Orchestration Specification for Cloud Applications version 1.0, 2013.

[100] OASIS. tosca-primer-v1.0-cnd01, 2013.

[101] OASIS. TOSCA Simple Profile in YAML Version 1 . 0 OASIS Standard. (December), 2016.

[102] T Oinn, M Addis, J Ferris, D Marvin, M Senger, M Greenwood, T Carver, K Glover, M. R Pocock, A Wipat, and P Li. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

[103] OMG. Deployment and Configuration of Component-based Distributed Applications Specification, 2006.

[104] OMG. OMG Unified Modeling Language TM (OMG UML), Superstructure, 2011.

[105] W. R Otte, A Gokhale, and D. C Schmidt. Efficient and deterministic application deployment in component-based enterprise distributed real-time and embedded systems. *Information and Software Technology*, 55(2):475–488, February 2013.

[106] W. R Otte, D. C Schmidt, and A Gokhale. Towards an Adaptive Deployment and Cofiguration Framework for Component-based Distributed Systems. In *Proceedings of the 9th Workshop on Adaptive and Reflective Middleware (ARM '10), Bengaluru*, Bengaluru, 2011.

[107] 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*, pages 1037–1040. IEEE, June 2015.

[108] R Qasha, J Cała, and P Watson. A Framework for Scientific Workflow Reproducibility in the Cloud. In *2016 IEEE 12th International Conference on eScience*, number October, 2016.

[109] R Qasha, J Cała, and P Watson. Dynamic Deployment of Scientific Workflows in the Cloud using Container Virtualization. In *2016 8th IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2016*, 2016.

[110] D. D Roure, P Missier, J Manuel, K. M Hettne, G Klyne, and C Goble. Towards the Preservation of Scientific Workflows. *iPress 2011*, 2011.

[111] J Ruiz, J Duenas, and F Cuadrado. Model-based context-aware deployment of distributed systems. *IEEE Communications Magazine*, 47(6):164–171, June 2009.

[112] B Russell. Kvm and docker lxc benchmarking with openstack. Technical report, IBM Global Technology Services, 2015.

[113] I Santana-Perez, R. F da Silva, M Rynge, E Deelman, M. S Pérez-Hernández, and O Corcho. Reproducibility of execution environments in computational science using semantics and clouds. *Future Generation Computer Systems*, 67(Supplement C):354 – 367, 2017.

[114] I Santana-Perez and M. S Pérez-Hernández. Towards Reproducibility in Scientific Workflows: An Infrastructure-Based Approach. *Scientific Programming*, 2015(4):1–11, 2015.

[115] 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.

[116] M. J Scheepers. Virtualization and Containerization of Application Infrastructure : A Comparison. *21st Twente Student Conference on IT*, pages 1–7, 2014.

[117] O Sefraoui, M Aissaoui, and M Eleuldj. Openstack: toward an open-source solution for cloud computing. *International Journal of Computer Applications*, 55(3), 2012.

[118] 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.*, 41:275–287, 2007.

[119] O Spjuth, M Dahlö, F Haziza, A Kallio, E Korpelainen, and E Bongcam-Rudloff. BioImg.org: A Catalog of Virtual Machine Images for the Life Sciences. *Bioinformatics and Biology Insights*, (Vmi):125, September 2015.

[120] O Standard. Topology and Orchestration Specification for Cloud Applications version 1.0, 2013.

[121] V Stodden, F Leisch, and R. D Peng. *Implementing reproducible research*. CRC Press, 2014.

[122] C Sun, L He, Q Wang, and R Willenborg. Simplifying Service Deployment with Virtual Appliances. In *2008 IEEE International Conference on Services Computing*, pages 265–272. IEEE Computer Society, July 2008.

[123] D Talia. Workflow Systems for Science: Concepts and Tools. *ISRN Software Engineering*, 2013:1–15, 2013.

[124] V Talwar and D Milojicic. Approaches for Service Deployment. *IEEE Internet Computing*, 9(2):70–80, March 2005.

[125] I. J Taylor, E Deelman, D. B Gannon, and M Shields. *Workflows for e-Science: Scientific Workflows for Grids*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[126] D Thain, T Tannenbaum, and M Livny. Distributed computing in practice: the Condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, February 2005.

[127] The OASIS Committee. Web Services Business Process Execution Language (WS-BPEL) Version 2.0, 2007.

[128] A Tosatto, P Ruiu, and A Attanasio. Container-Based Orchestration in Cloud: State of the Art and Challenges. *Proceedings - 2015 9th International Conference on Complex, Intelligent, and Software Intensive Systems, CISIS 2015*, pages 70–75, 2015.

[129] W. M. P Van Der Aalst. The Application of Petri Nets To Workflow Management. *Journal of Circuits, Systems and Computers*, 08(01):21–66, 1998.

[130] S van der Burg and E Dolstra. Automated Deployment of a Heterogeneous Service-Oriented System. In *2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 183–190. IEEE, September 2010.

[131] L. M Vaquero, L Rodero-Merino, J Caceres, and M Lindner. A break in the clouds. *ACM SIGCOMM Computer Communication Review*, 39(1):50, 2009.

[132] S. J Vaughan-nichols. New approach to virtualization is a lightweight. *Computer*, 39(11):12–14, Nov 2006.

[133] K Vukojevic-haupt, S. G Sáez, F Haupt, D Karastoyanova, and F Leymann. A Middleware-centric Optimization Approach for the Automated Provisioning of Services in the Cloud. In *the 2015 IEEE 7th International Conference on Cloud Computing Technology and Science, CloudCom 2015*, pages 174–179, Vancouver, Canada, 2015. IEEE.

[134] W3C Member Submission. Installable Unit Deployment Descriptor Specification Version 1.0, 2004.

[135] B Walters. Vmware virtual platform. *Linux J.*, 1999(63es), July 1999.

[136] J Wang, P Korambath, I Altintas, J Davis, and D Crawl. Workflow as a Service in the Cloud: Architecture and Scheduling Algorithms. *Procedia Computer Science*, 29:546–556, 2014.

[137] Z Wen, J Cala, P Watson, and A Romanovsky. Cost effective, reliable and secure workflow deployment over federated clouds. *IEEE Transactions on Services Computing*, PP(99):1–1, 2016.

[138] J Wettinger, V Andrikopoulos, F Leymann, and S Strauch. Middleware-oriented Deployment Automation for Cloud Applications. *IEEE Transactions on Cloud Computing*, pages 1–1, 2016.

[139] J Wettinger, V Andrikopoulos, S Strauch, and F Leymann. Characterizing and evaluating different deployment approaches for cloud applications. In *Proceedings of the IEEE International Conference on Cloud Engineering (IEEE IC2E 2014)*, pages 205–214. IEEE Computer Society, 2014.

[140] 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.

[141] S Woodman, H Hiden, P Watson, and P Missier. Achieving reproducibility by combining provenance with service and workflow versioning. In *Proceedings of the 6th Workshop on Workflows in Support of Large-scale Science*, WORKS '11, pages 127–136, New York, NY, USA, 2011. ACM.

[142] L Youseff, M Butrico, and D. D Silva. Toward a unified ontology of cloud computing. In *2008 Grid Computing Environments Workshop*, pages 1–10, Nov 2008.

[143] T Zhang, Z Du, Y Chen, X Ji, and X Wang. Typical Virtual Appliances: An optimized mechanism for virtual appliances provisioning and management. *Journal of Systems and Software*, 84(3):377–387, March 2011.

[144] Y Zhang, Y Li, and W Zheng. Automatic software deployment using user-level virtualization for cloud-computing. *Future Generation Computer Systems*, 29(1):323–329, January 2013.

[145] 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*, pages 1–9. IEEE, October 2012.

[146] Y Zhao, X Fei, I Raicu, and S Lu. Opportunities and challenges in running scientific workflows on the cloud. In *2011 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, pages 455–462, Oct 2011.

[147] Y Zhao, Y Li, S Lu, I Raicu, and C Lin. Devising a cloud scientific workflow platform for big data. In *2014 IEEE World Congress on Services*, pages 393–401, June 2014.

[148] Y Zhao, Y Li, I Raicu, S Lu, C Lin, Y Zhang, W Tian, and R Xue. A service framework for scientific workflow management in the cloud. *IEEE Transactions on Services Computing*, 8(6):930–944, Nov 2015.

[149] 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*, (1), nov 2014.

[150] A. C Zhou, B He, and C Liu. Monetary cost optimizations for hosting workflow-as-a-service in iaas clouds. *IEEE Trans. Cloud Comput.*, 4(1):34–48, January 2016.