UNIVERSITY OF BATH

University of Bath

A Light-Weight PaaS Cloud Framework for Scientific Applications

Kewei Duan[1]⋆, Julian Padget[1], and H. Alicia Kim[2]

[1] Department of Computer Science, University of Bath
{k.duan,j.a.padget}@bath.ac.uk
[2] Department of Mechanical Engineering, University of Bath
h.a.kim@bath.ac.uk

Abstract. A significant trend in science research for at least the past decade has been the increasing uptake of computational techniques (modelling) for in-silico experimentation, which is trickling down from the grand challenges that require capability computing to smaller-scale problems suited to capacity computing. Such virtual experiments also establish an opportunity for collaboration at a distance. At the same time, the development of web service and cloud technology, is providing a potential platform to support these activities. The problem on which we focus is the technical hurdles for users without detailed knowledge of such mechanisms – in a word, 'accessibility' – specifically: (i) the heavy weight and diversity of infrastructures that inhibits shareability and collaboration between services, (ii) the relatively complicated processes associated with deployment and management of web services for non-disciplinary specialists, and (iii) the relative technical difficulty in packaging the legacy software that encapsulates key discipline knowledge for web-service environments. In this paper, we describe a light-weight PaaS cloud framework based on REST to address the above issues. The framework provides a PaaS model that allows users to deploy REST services from the desktop on to computing infrastructure without modification or recompilation, utilizing legacy applications developed for the command-line. A behind-the-scenes facility provides asynchronous distributed staging of data (built directly on HTTP and REST). We describe the framework, comprising the service factory, data staging services and the desktop file manager overlay for service deployment, and present experimental results regarding: (i) the improvement in turnaround time from the data staging service, and (ii) the evaluation of usefulness and usability of the framework through case studies in image processing and in multi-disciplinary optimization.

1 Introduction

With the increasing uptake of computational techniques for in-silico experimentation, scientists seek capacity computing power along with the means to collaborate at a distance. Web services in principle provide a convenient means to publish and share computational representations of domain-specific knowledge, while grid computing has initially delivered the infrastructure for capability scientific computing[1–3]. More recently, cloud computing, which can be seen as an evolution of the latter, offers a more

---

⋆ student author

accessible and flexible provisioning of capacity computing, that renders the usability issues around complex infrastructure largely invisible to end users. It also shows benefits for scientific applications in a wide range of domains[4–6]. Various approaches and tools have been developed that can deploy scientific applications as cloud resources. However, there are still hurdles for scientists who have limited technical knowledge of cloud computing infrastructure and of the use of new technology in scientific applications. We identify these hurdles as: (i) the heavy weight and diversity of infrastructures that inhibits shareability and collaboration among distributed services, (ii) the relatively complicated processes associated with deployment and management of web services for non-discipline specialists, (iii) the relative technical difficulty in packaging the legacy software that encapsulates key discipline knowledge for web-service environments.

The aforementioned hurdles are determined by the nature of the end-user-scientist and the resources that need to be deployed in the cloud. Most scientists who have limited knowledge on web services or cloud infrastructure may need to face the reality that they need to learn new programming languages or system administrative skills for the purpose to build scientific applications in the cloud or as web services. For example, an engineer normally has the skill to develop desktop applications based on Fortran or Matlab, but rarely has knowledge of or experience of web application development based on languages like Java or Python. On the other hand, with years of development, numerous legacy codes and programs in which real domain-specific knowledge resides, may face the predicament that a new round of coding and translating work is needed or they simply lose the ability to be re-developed because of the lack of source codes, documents or language supports[3].

Our REST based PaaS cloud framework lowers the barriers by providing a set of GUI based client tools and a set of REST web services which serve as both portal for service deployment and service execution by following the PaaS service model[7]. This framework is able to deploy legacy codes and command-line programs as REST web service, which can cover a wide range of languages and tools, such as C/C++, Fortran, Matlab, Python, Unix shell, JAVA, and even some engineering design optimization frameworks, such as OpenMDAO[8] and Dakota[9]. Furthermore, because the framework is entirely built on the REST architecture, it can be directly accessed from a wide range of programming languages (such as a command line scripts/applications) or a generic workflow management system (such as Taverna[10], see section 4) without extra library support and tools. The services are made into as web applications, based on easily obtainable, free, open source tools, such as Apache-Tomcat and MySQL. Embedded within the framework is a distributed data-flow mechanism, that can enhance data staging performance in the execution of composite services. With the support of the desktop GUI tool, inexperienced users can learn about and use web services. We demonstrate the framework operating both in the context of a private server and the Amazon EC2 service, in order to show compatibility with both private and public cloud provisioning. Hence, we believe it should be readily deployable on top of other IaaS services with little change.

---

[3] In the worst case, only a binary of the program may exist, which happens to be executable due to backwards hardware compatibility.

The primary technical contributions of the paper are: (i) the design of a RESTful framework for the deployment of legacy codes through a *service factory* facility, (ii) an architecture for the execution of those services, in which data services are supplied by an asynchronous data-flow mechanism providing *Data as a Service* and control can be provided by existing workflow engines, such as Taverna, and (iii) a *desktop GUI* and file system overlay to provide the interface for service management. Complementary to these is the social contribution, of providing access to web service functions, cloud computing infrastructure and user-controlled means for sharing the scientific knowledge embedded in computational resources (software). These aspects have been evaluated, using recognized HCI practices[11–13] on the one hand through participatory exercises and surveys (usability) and on the other through two case studies (usefulness).

The rest of the paper is structured as follows. In Section 2, we discuss the challenges of migrating scientific applications to cloud and related work. Section 3 introduces our framework and the solutions ew propose to meet those challenges. Section 4 evaluates the framework in respect of three issues: (i) performance, (ii) user-based experiments, and (iii) (two) case studies. Lastly, Section 5 presents conclusions and future work.

2 Related Work

Cloud computing is commonly categorized into three service models[7] known as {Infrastructure, Platform, Software} as a Service (IaaS, PaaS and SaaS, respectively), of which PaaS is the service model that provides the consumer with the capability to deploy consumer-created or acquired applications onto infrastructure, thus creating an instance of a service. Applications can be created by using programming languages, libraries, services, and tools supported by the service provider. The other two models are Software-as-a-Service (SaaS) and Infrastructure-as-a-Service (IaaS). In contrast with PaaS, Saas model provides applications as scalable services that can be customized or composed into other applications, while, IaaS needs more management work from users on computing infrastructure, such as operating system, file-based storage, firewalls, etc. This paper focuses on the effective use of PaaS cloud for scientific applications. Based on this PaaS cloud, applications will appear as web services in a SaaS service model for public invocation. There are several generic PaaS platforms like Google APP Engine [14] and Heroku [15], both of which provide the means for users to deploy web applications on the providers' public cloud infrastructure. However, both of them work via programming language APIs. For the purpose of deploying an application into their infrastructures, users must either write applications in specific languages or modify original codes in those languages.

Toolkits such as Soaplab[16], Opal[17] and Generic Factory Service (GFac)[2] wrap command-line applications for service deployment. Users can use them to describe the command-line and parameters to create services. They differ from our framework in several ways:

1. We use a PaaS cloud model to provide the function of service deployment as web service, which allows hot-plug style program uploading and deployment. The above assume programs have been installed on the server and work as local tools on a server that needs to be set up and configured every time a new service is deployed.

2. We consider the deployment of web service in a broader context, assuming they will be composed, while a data staging mechanism is provided to assist in the effective composition of services. The above tools do not consider data communication as their concern, which can in the worst case result in centralized data transfer, when deployed as web services.

3. We provide a desktop GUI tool for clients to deploy web services based on command-line programs. This is significantly easier than the description languages adopted in these tools ("Ajax Command Definition" in Soaplab, "serviceMap" in GFac and "Metadata" in Opal), all of which require learning and are typically written by hand, creating a hurdle for learning and use.

The Generic Worker framework [18] has similar goals to our framework: it provide PaaS service based on Microsoft's Azure Cloud platform [19]. Services can be deployed by the client using command-line tools. They also adopt a distributed data transfer mechanism for performance enhancement. However, their services are tightly connected to Azure service elements, such as Azure's REST web service API and the Azure blob store. Our framework should be deployable in any private cloud or any popular public cloud based as it is on a set of open-source tools. The data can also reside in a range of cloud computing storage, such as Dropbox, Ubuntu one, or SpiderOak, for example. We also note that data elements in our framework are transferred and stored without additional mark-up. To facilitate the delivery of the right data at the right time in the right place, we provide a data-flow style Data-as-a-Service (DaaS) mechanism, called Datapool, that keeps all the data in their original format (ie., no encoding, no wrapping) and provides for asynchronous data transfer between services (described in detail in Section 3).

3 A PaaS Framework for Scientific Applications

In this section, we describe our framework and how we believe it addresses the issues raised by the hurdles we identified earlier. We approach these issues from three directions: (i) service deployment, (ii) service invocation and execution, (iii) data staging mechanism. The framework is composed of a set of RESTful web services, which are deployed as web applications in Apache-Tomcat, and a client GUI tool for service management. First, we explain how scientific applications are uploaded and deployed dynamically as web service through our client GUI tool, then we describe invocation and execution as REST web services through APIs or generic workflow management system. We finally describe how our framework provides distributed data staging between client and services, again based on RESTful web services.

3.1 Service Deployment

Scientific applications must be uploaded and registered with the framework before they are available for invocation and execution in the cloud. There are three tasks at this stage: (i) the first is to upload and store the application and its dependencies in the cloud repository, (ii) the second is to write and upload the description of the application
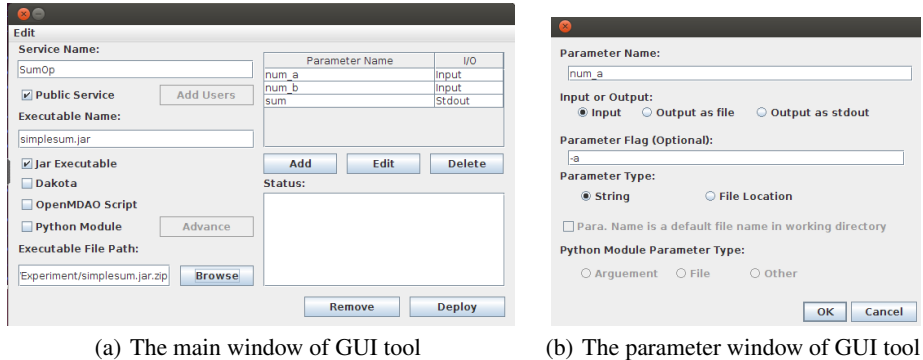
(a) The main window of GUI tool   (b) The parameter window of GUI tool
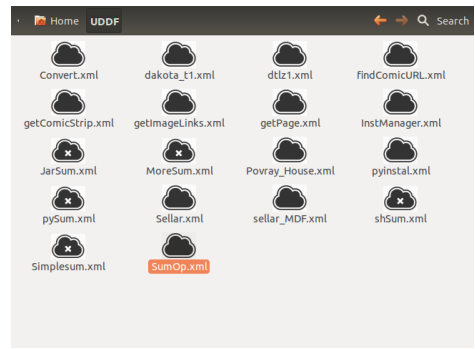
Fig. 1. Windows of GUI tool

Fig. 2. Local folder for service description

to cloud for subsequent configuration and deployment, (iii) the final task is the configuration and uploading of authorization information such as owner information and the access control list for the service. These tasks can be all performed through the client GUI tool.

Figure 1(a) shows the main window of the GUI tool. This tool is set up to connect with the management service in cloud through a URI with user authentication information. For the first application uploading task, the user has to pack all the binary and dependencies in a self-contained folder as a compressed file and upload it cloud side through the management service. This tool can be started from the menu when the user right-clicks on the compressed file[4]. In the interface of the GUI tool, a browse button at the bottom is used to indicate the file location. In this case, a Java executable which has two inputs and one output is uploaded. The Java runtime is a special case that can be specified by ticking "Jar executable".

---

[4] Thanks to integration with the file manager. Although, in this case, the integration is with the Nautilus file manager on Ubuntu, such overlays are common interface extensions on other operating systems, so we view this as a generic technique.

Figure 1(b) shows the parameter window of the GUI tool, which is used to describe the input/output parameters of the program. The framework can deal with a wide range of I/O types for command-line programs, including string value, file path, argument flag, as well as standard input/output stream and default output files in the current 'working' directory. For example, in this case, the command line for this Java executable is `java -jar simplesum.jar -a {num_a} -b {num_b}`. Output is to the standard output stream. Figure 1(b) contains the configuration for the first input `num_a`. The other information such as service name can be defined in the main window.

In Figure 1(a), there is also an option to say whether this service is public, meaning it can be invoked by anyone, or not, in which case it can only be accessed by specified users. The last task, regarding authorization information configuration, can be carried out by clicking the "Add Users" button. After all the information is entered via this tool, the deployment task is completed by clicking the "Deploy" button.

In this deployment process, the description is contained in a XML file which is automatically generated by the tool. One copy is uploaded to cloud repository. The other copy is stored in a local designated location. Figure 2 shows the folder contains all the descriptions. Users can operate on them by starting the GUI tool from the right-click menu, to access operations for remove, modify and redeploy. The description of any service that is removed is kept in the folder, identified by a cloud icon with a cross, for possible future redeployment.

3.2   Service Invocation and Execution

All the services, which include data services and application services, are provided as RESTful web services. In recent years, the REST architectural style[20] and REST-compliant Web services[21] have emerged and the approach has rapidly gained popularity due to its flexibility and simplicity. At the same time, to parallel the development of Service-Oriented Architecture (SOA) alongside arbitrary web services[21], we can also observe the development of the concept of the Resource-Oriented Architecture (ROA). ROA defines a specific set of web-based system design principles derived from the implementation of the REST architecture, namely addressability, statelessness, connectedness and a uniform interface. Thus, resources in a RESTful system can be accessed universally through uniform methods based on the HTTP protocol. In contrast, in the arbitrary web services based solutions[16, 17, 2], each application designer is required to define a new and arbitrary protocol comprising vocabulary of nouns and verbs that is usually overlaid on the HTTP protocol. By working directly with HTTP, REST has fewer protocol layers, which reduces the overheads in data preparation and subsequent processing computation. Pautasso et al.[22] conclude that for the objective of designing IT architectures, REST is a better choice in the context of a cooperative environment, because it is intrinsically loosely coupled.

Table 1 shows all the URIs of the two types of services. Datapool services are the services for I/O data items manipulation (uploading, retrieval, etc.). Application services include the services for application service deployment and execution. Uniform methods based on HTTP protocol are allocated to each URI for specific operation. For example, the first and third service in the Application services list have the same URI,

	Methods	URIs
Datapool Services	PUT	http://.../datapool/{Datapool_Name}/{Data_Object_Name}
	PUT	http://.../datapool/{Datapool_Name}?DO_URI={Data_Object_URI}
	GET	http://.../datapool/{Datapool_Name}/{Data_Object_Name}
	GET	http://.../datapool/{Datapool_Name}
	DELETE	http://.../datapool/{Datapool_Name}/{Data_Object_Name}
	DELETE	http://.../datapool/{Datapool_Name}
Application Services	PUT	http://.../APP_service/{Service_Name}
	GET	http://.../APP_service/{Service_Name}?DP_URI={Datapool_URI}
	DELETE	http://.../APP_service/{Service_Name}
	GET	http://.../APP_service/Service_Info/{Service_Name}

Table 1. URIs of Datapool and Application Services

which represents one application resource. The PUT method denotes a service deployment operation, while DELETE denotes a service removal operation. These services also support a role-based authorization system so that only an authenticated and authorized user can access those services. Authentication is carried out over HTTP and communication can be further encrypted and secured by HTTPS through the Transport Layer Security (TLS) protocol. In Section 3.1, we describe the means to specify the authorization permissions for a given service.

Datapools are special components that bring benefits to service execution. Each datapool represents a collection of data items each with a unique URI. Multiple Datapool instances can be generated and customized through the Datapool service by the user. Each data item inside one Datapool is allocated with unique URI as well. Only the creator of each Datapool can access the content, which is guaranteed by the authorization mechanism. There are two advantages to organizing data in this way. First, because all the data items and data collection are directly allocated with URIs, they are all web resources that can be re-accessed through HTTP at any time rather than merely a data stream in the form of extra layer of XML or other structure. Therefore, each data item can also be transferred and kept in their original textual or binary format. Second, in the execution of an application service, the URI of one Datapool that contains all the input data is provided to the service. The application will pull the necessary data automatically from the provided local or remote Datapool. In this way, the interfaces are unified for different application services in the form of a URI, of which the Datapool URI is a constituent as a query string. The second URI in the application services list in 1 shows the unified format.

Figure 3 depicts an execution process example based on a Datapool service with two application services. In this scenario, we assume the two application services are located in different servers forming one sequential execution process. There are two corresponding Datapool services on each server as well. SERVICE 1 needs INPUT1 and INPUT2 as input files and SERVICE 2 needs output from SERVICE 1 and INPUT3 as input files. INPUT1–3 are the original data objects uploaded from the client side through the PUT method of one Datapool service. OUTPUTS COLLECTION 1, which contains intermediate data objects generated by SERVICE 1, does not need to be transferred back to the client. Data transmission only happens between Datapools in
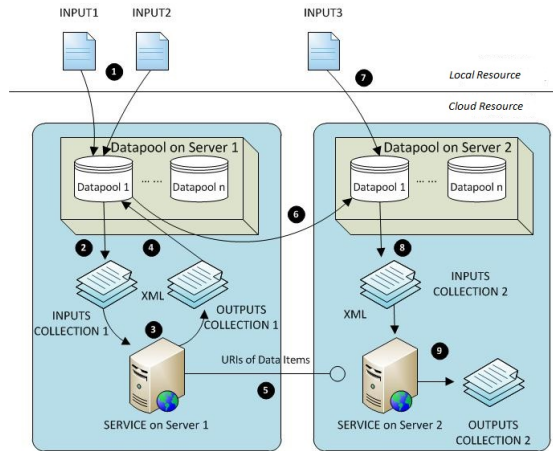
Fig. 3. An execution process example based on Datapool service

each SERVER. Data elements are transferred from server to server under the control of the client side. Therefore, in Figure 3, data objects can be pulled from SERVER 1 to SERVER 2 in step 6. Meanwhile, step 7 can be carried out concurrently with step 6 as well as step 1. The actual execution of the two application services happens at step 5 and step 9, respectively. The URIs of the corresponding Datapools are supplied in these steps.

3.3 The Data Staging Mechanism

Data staging and how to control it are not new problems. Already in 1997 [23], adopted the idea of distributed data-flows in a service composition framework to improve data transfer performance, as did also [24] some years later. Similar ideas are embodied in some distributed program execution engines, such as [25, 26], to overcome the bottle-neck of data transfers. Meanwhile, several workflow management systems took up a peer-to-peer style mechanism for intermediate data movement[27–29]. Although there are differences in detail between the various aforementioned solutions, there is one common aspect, namely the use of a private – by which we mean internal, or closed – mechanism (functions are exposed by a set of developer defined specific interfaces and operations) to handle data transfer. A further point in common is the need for address-ability: in each case the data objects are assigned some unique label that allows them to be accessed from any location on the network that is participating in the enactment process. These works inspired our data staging mechanism based on cloud resources and REST.

We can make two quite obvious remarks about dataflows among multiple services: (i) for a given service invocation, the dataflow rarely involves the client or central con-troller, which means that dataflows can (normally) be distributed, and (ii) it is not un-common that the necessary data objects (inputs) may come from different sources, sug-
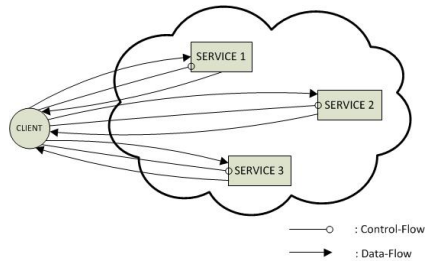
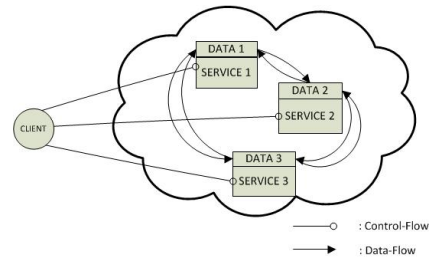Fig. 4. Centralized Data-Flows in Web Services Compostion

Fig. 5. Distributed Data-Flows in Web Services Composition

gesting that data transfers can be initiated asynchronously before the actual execution of a service. These constitute the properties our data staging mechanism needs to satisfy.

In our data staging mechanism, data collection is organized by Datapool. Each Datapool contains data resources, each of which has their own unique universal address (URI). This is the basis on which the data involved can be accessed individually and transferred separately. In this section, we discuss this mechanism from both a distributed and an asynchronous data transfer perspective.

Distributed data transfer Figures 4 and 5 illustrate the essential difference between a centralized and a distributed mechanism for data transfer. Figure 4 shows that both control-flow and data-flow are centrally coordinated for each Web service invocation. There is a high risk that the client or central controller becomes a bottleneck for data communication among computation components. In Figure 5, the data-flows are distributed among Web services directly rather than passing through a central controller, which also allows for the concurrent transfer of data items from different resources. This process is also demonstrated in the example of Section 3.2. The client can also obtain the complete set of data objects whenever it is desired. Hence, each service provider takes care of the task of data storage instead of the client. Furthermore, each data object has the capability to be identified and accessed universally through the Internet based on URI.

Asynchronous data transfer The asynchronous data transfer process is illustrated in Figure 6, where there are three data objects from three different web services that need to be transferred to another service as inputs through three different connections. For diagrammatic convenience, we make the not entirely realistic assumption that in two situations, the same data object transfer takes the same time; thus the length of each bar represents time. Under synchronous data transfer, because the data references are controlled through the client, data transfer only starts when the last service finishes and the next service invocation happens. However, in the asynchronous method, the transfers start asynchronously when each previous service finishes. The transfers are not synchronized with the invocation of next service, rather data elements are transferred and stored in the 'next' Datapool in advance. From Figure 6, we can clearly see that the asynchronous method may be able to bring about an earlier completion of the whole data transfer process. Both distributed and asynchronous data transfer are ensured by the Datapool web service.
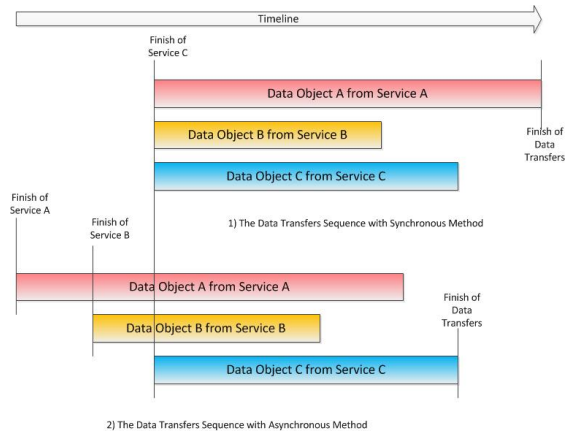
Fig. 6. Comparison of two different data transfers mechanisms

Because it is a REST service, addressability enables data items to be accessed individually and universally across the network at any time.

4 Evaluation

4.1 Experiment on Usefulness and Usability

A formal experiment with an after-experiment survey is carried out to collect evidence for the usefulness of the GUI tool-based service management mechanism. The objective here is assess usage of the tool for users who do not have any experience of building or deploying web services. A secondary aim is to collect evidence for the usability of the GUI. In this experiment, four programs are provided to participants. Three of them have two inputs and one output, and are written in Java, Python and Unix shell, respectively. The other has three inputs and two outputs and is written in Python. The experiment has four stages: (i) a 3–5 minute training stage, which includes a tutorial video and question time, (ii) three simple programs are provided to participants to deploy in an order that they decide, while the operation time is recorded, (iii) a more complicated program for which deployment time is also recorded, and (iv) completing the survey.

Figure 7 shows the average time and full time range for deployment operations based on data collected from 9 participants. We note that none of the subjects claimed any prior experience of building or deploying web services.

In a question about their subjective views on simplicity with 5-point scales from very easy (1) to very difficult (5), 2 out 9 said very easy (1), and the rest said easy (2). All the participants successfully deployed web services in around 2 minutes. In the randomly ordered simpler cases, it can be noticed that there is a significant fall in the time taken. It also can be noticed that after three test cases, the time taken for the more difficult case is less than the first of the simple ones. The objective evidence obtained from this experiment is that the GUI based mechanism is easy to learn and use.
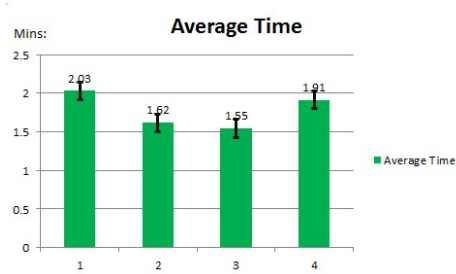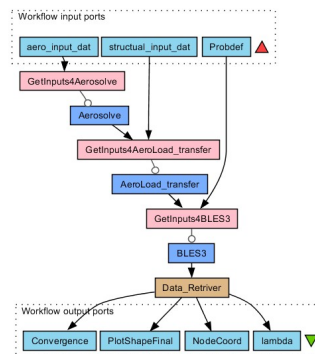
Fig. 7. The average time of deployment operations

Fig. 8. The wing structure optimization process built in Taverna

4.2 Case Studies

Image Processing Workflow In this workflow, the binaries for PovRay[30] and ImageMagick[31] are installed on the server-side of the framework. PovRay is a ray tracing program to draw 3-D image from scene description that is written in the POV description language. ImageMagick is a software suite to create, edit, compose, or convert images. In this case, we create a workflow to output a 3-D image in `png` format starting from a POV description as input, and then convert it to jpg format by ImageMagick. Both of their execution process are written as Unix shell scripts. They are all deployed through the GUI tool as web services. In the deployment process, PovRay dependency files in the format of `inc` can be compressed and uploaded to build the web service. The workflow contains two Datapool services and two Application services. They are invoked from the client-side by an executable script written in Python, which supports the invocation of RESTful web services. The `png` file is an intermediate data object, which is not transferred back to the client. The Datapool service for ImageMagick receives this image as a URI reference (step 6 in Figure 3).

This case study serves to demonstrate how the binary versions of two command-line programs can be turned into web services and then invoked from a command-line program written in Python.

Multi-Disciplinary Optimization (MDO) Workflow Multi-disciplinary design optimization (MDO) is a field of engineering that uses (multi-objective) optimization methods to solve design problems combining a number of disciplines. For the purpose of demonstrating multi-disciplinary design optimization process as a web services composition, we use the Taverna workflow management system [10] to carry out the tasks of composition, execution and monitoring, as in our previous work [32, 33]. The services composition can work in a distributed data staging mechanism based on our framework, although the intermediate data movement of Taverna is categorized as centralized style[34, 35]. Figure 8 shows a screenshot of the services composition design example, which serves to optimize the internal stiffness distribution of a typical aircraft wing under coupled aerodynamics and structural consideration.

In Figure 8, all the dark blue boxes (Aerosolve, AeroLoad_transfer, BLES3) are services deployed based on three command line programs, written in Fortran or C. All the pink boxes (GetInputs4Aerosolve, GetInputs4AeroLoad_transfer, GetInputs-BLES3) are the Datapool services. The input ports (light blue boxes) built in Taverna are located at the top of Figure 8, and output ports (light blue boxes) at the bottom. One local service, the Data Extractor, retrieves the data based on the URIs returned by the last application service.

Our framework can also deploy legacy MDO workflows based on existing MDO frameworks like OpenMDAO[8] and Dakota[9]. OpenMDAO is based on Python an a workflow is written as an executable python script. With the support of the OpenM-DAO runtime installed in server, the process of deployment can be done as easily as any other command-line programs. Dakota has a different execution approach in that the workflow is defined as a input file, which can be executed by the Dakota runtime. With the Dakota runtime installed in server, the workflow can be executed as a web service by simply uploading the input file through the Datapool service. The options for deploying OpenMDAO and Dakota services are included in the GUI interface, which is demonstrated in Figure 1(a).

This case study primarily serves to show how a popular workflow engine can enact a workflow whose services are the result of our deployment mechanism, thus enabling composition at a programmatic level and the sharing of computational knowledge embedded in software.

4.3   Comparison of Data Staging Performance

5   Conclusion and Future Work

In this paper, we have presented evidence for the benefits arising from our light-weight PaaS framework for the deployment and execution of scientific application in the cloud. With our GUI based deployment mechanism, the technical barriers are lowered for non-specialist usage of web services and cloud resources. With the use of REST web service interfaces, services can be accessed from multiple client tools and programming languages without additional protocol layers or dependencies, such as proprietary client tools or libraries. The framework reduces the effort for users to turn legacy codes and programs into web services and hence collaborate with each other. We also demonstrate its ability to work with the generic workflow workbench Taverna[10]. The distributed and asynchronous data staging mechanism helps reduce end-to-end times by hiding the costs of data staging between services as well as between client and service. This paper also evaluates the usefulness and usability of the framework through a simple user study and case studies, showing how different types of legacy programs and tools can cooperate seamlessly in workflow with the support of our framework.

In future work, we need to address support for the construction and deployment of composite services: one approach we have explored as proof-of-concept, is to treat a Taverna workflow as a service to be executed, where the workflow description is the data and the program is the enactment engine. Similar functionality should also be achievable with Kepler. A more serious issue however, is the dependence on actual services, meaning there is a reliance on a service provided at a specific URL, as against

a specification of a service by, say, its profile (in OWL-S terminology), and the late binding identification of suitable available candidate services close to enactment time. A preliminary effort in this direction appears in [36], based on a matchmaker that assumes WSDL format service descriptions, but a fresh approach that takes advantage of REST seems desirable when this is revisited. Hence, we hope this framework will allow more users to build their services and participate in service composition. Finally, we propose to take advantage of the availability of capacity computing facilities to support speculative enactment of services, following the design set out in [37].

References

1. Gannon, D., Ananthakrishnan, R., Krishnan, S., Govindaraju, M., Ramakrishnan, L., Slominski, A. In: Grid Web Services and Application Factories. John Wiley & Sons, Ltd (2003) 251–264
2. Kandaswamy, G., Fang, L., Huang, Y., Shirasuna, S., Marru, S., Gannon, D.: Building web services for scientific grid applications. IBM Journal of Research and Development **50**(2.3) (2006) 249–260
3. Sneed, H.: Integrating legacy software into a service oriented architecture. In: Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on. (2006) 11 pp.–14
4. Gorder, P.F.: Coming soon: Research in a cloud. Computing in Science and Engineering **10**(6) (2008) 6–10
5. Sullivan, F.: Guest editors introduction: Cloud computing for the sciences. Computing in Science & Engineering **11** (2009) 10
6. Rehr, J.J., Vila, F.D., Gardner, J.P., Svec, L., Prange, M.: Scientific computing in the cloud. Computing in Science & Engineering **12**(3) (2010) 34–43
7. Mell, P., Grance, T.: The NIST definition of cloud computing. National Institute of Standards and Technology, Special Publication 800-145 (2011) [Online; accessed 02-May-2013].
8. NASA Glenn Research Center: OpenMDAO. http://openmdao.org [Online; accessed 08-March-2013].
9. Sandia National Laboratories: The DAKOTA Project. http://dakota.sandia.gov/ [Online; accessed 08-March-2013].
10. School of Computer Science, University of Manchester: Taverna. http://www.taverna.org.uk/
11. Landauer, T.K.: The trouble with computers: Usefulness, usability, and productivity. Volume 21. Taylor & Francis (1995)
12. Kitchenham, B.A.: Evaluating software engineering methods and tool part 1: The evaluation context and evaluation methods. ACM SIGSOFT Software Engineering Notes **21**(1) (1996) 11–14
13. Moody, D.L.: Theoretical and practical issues in evaluating the quality of conceptual models: current state and future directions. Data & Knowledge Engineering **55**(3) (2005) 243–276
14. Google: Google app engine. http://developers.google.com/appengine/ (2008) [Online; accessed 02-May-2013].
15. Lindenbaum, J., Wiggins, A., Henry, O.: Heroku. http://www.heroku.com (2008) [Online; accessed 02-May-2013].
16. Senger, M., Rice, P., Bleasby, A., Oinn, T., Uludag, M.: Soaplab2: more reliable Sesame door to bioinformatics programs. In: 9th Annual Bioinformatics Open Source Conf. (2008)

17. Krishnan, S., Clementi, L., Ren, J., Papadopoulos, P., Li, W.: Design and evaluation of opal2: A toolkit for scientific software as a service. In: Services - I, 2009 World Conference on. (2009) 709–716
18. Simmhan, Y., van Ingen, C., Subramanian, G., Li, J.: Bridging the gap between desktop and the cloud for escience applications. In: Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on. (2010) 474–481
19. Microsoft: Windows Azure: Microsoft's Cloud Platform. http://www.azure.com/ [Online; accessed 08-May-2013].
20. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. PhD thesis, University of California, Irvine (2000)
21. W3C: Web Services Architecture. http://www.w3.org/TR/ws-arch/#relwwwrest [Online; accessed 08-August-2011].
22. Pautasso, C., Wilde, E.: Why is the Web Loosely Coupled? A Multi-Faceted Metric for Service Design. In: Proc. of the 18th International World Wide Web Conference (WWW2009), Madrid, Spain (April 2009) 911–920
23. Alonso, G., Reinwald, B., Mohan, C.: Distributed data management in workflow environments. In: Research Issues in Data Engineering, 1997. Proceedings. Seventh International Workshop on. (apr 1997) 82 –90
24. Liu, D., Peng, J., Wiederhold, G., Sriram, R.D., Aruthor, C., Law, K.H., Law, K.H.: Composition of engineering web services with distributed data flows and computations (2005)
25. Murray, D.G., Schwarzkopf, M., Smowton, C., Smith, S., Madhavapeddy, A., Hand, S.: CIEL: a universal execution engine for distributed data-flow computing. In: Proceedings of the 8th USENIX conference on Networked systems design and implementation. NSDI'11, Berkeley, CA, USA, USENIX Association (2011) 9
26. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. In: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007. EuroSys '07, New York, NY, USA, ACM (2007) 59–72
27. UC Davis, UC Santa Barbara, and UC San Diego: Kepler project. https://kepler-project.org/
28. Cardiff University: Triana project. http://www.trianacode.org/
29. Cao, J., Jarvis, S., Saini, S., Nudd, G.: Gridflow: workflow management for grid computing. In: Cluster Computing and the Grid, 2003. Proceedings. CCGrid 2003. 3rd IEEE/ACM International Symposium on. (may 2003) 198 – 205
30. Persistence of Vision Raytracer Pty. Ltd.: Povray. http://www.povray.org/
31. ImageMagick Studio: Imagemagick. http://www.imagemagick.org
32. Duan, K., Seowy, Y.V., Kimz, H.A., Padget, J.: A Resource-Oriented Architecture for MDO Framework. In: Proceeding of 8th AIAA Multidisciplinary Design Optimization Specialist Conference. (2012)
33. Duan, K., Padget, J., Kim, H.A., Hosobe, H.: Composition of Engineering Web Services with Universal Distributed Data-Flows Framework based on ROA. In: Proceedings of the WS-REST 2012. (2012)
34. Yu, J., Buyya, R.: A taxonomy of workflow management systems for grid computing. Journal of Grid Computing **3** (2005) 171–200 10.1007/s10723-005-9010-8.
35. Yang, Y., Liu, K., Chen, J., Lignier, J., Jin, H.: Peer-to-peer based grid workflow runtime environment of swindew-g. In: Proceedings of the Third IEEE International Conference on e-Science and Grid Computing, Washington, DC, USA, IEEE Computer Society (2007) 51–58
36. Chapman, N., Ludwig, S., Naylor, W., Padget, J., Rana, O.: Matchmaking support for dynamic workflow composition. In: Proceedings of 3rd IEEE International Conference on eScience and Grid Computing, IEEE (December 2007) 371–378 DOI:10.1109/E-SCIENCE.2007.48.

37. Fukuta, N., Satoh, K., Yamaguchi, T.: Towards "kiga-kiku" services on speculative compu-
tation. In Yamaguchi, T., ed.: PAKM. Volume 5345 of Lecture Notes in Computer Science.,
Springer (2008) 256–267