A Light-Weight Framework for Bridge-Building from Desktop to Cloud

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 framework based on cloud and REST to address the above issues. The framework provides a 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

⋆ Student author.

A.R. Lomuscio et al. (Eds.): ICSOC 2013 Workshops, LNCS 8377, pp. 308–323, 2014.

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 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]. 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 them 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 of web services or cloud infrastructure may need to face the 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 support[1].

Our REST based light-weight 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]. In recent years, the REST architectural style[8] and REST-compliant Web services have emerged and the approach has rapidly gained popularity due to its flexibility and simplicity. Our framework is able to deploy legacy codes and command-line programs as RESTful services, which can support a wide range of languages and tools, such as C/C++, Fortran, Matlab, Python, Unix shell, JAVA, and some engineering design optimization frameworks, specifically OpenMDAO[9] and Dakota[10]. Furthermore, because the framework follows RESTful principles, 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[11], see section 4) without any additional library support or 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. Through the desktop GUI tool, inexperienced users can learn about, create 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

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

provisioning. Hence, we believe it should be readily deployable on top of other IaaS services with little change.

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[12, 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 proposed 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. Our aim is to provide access to cloud services so that regular users can deploy their own (command-line) applications as services, share them with others and utilise them in service workflows. We do this through the provision of a platform that provides: (i) deployment services, and (ii) data storage and transfer services.

This paper focuses on the use of cloud platform for science and engineering applications, in which the platform enables applications to appear as web services, creating a SaaS for public invocation. Our aim to provide a platform for users without sophisticated programming skills to be able deploy web services. 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. Other potential platforms – providing command-line interfaces – are: (i) CloudFoundry [16], which provides an open-source mechanism for application deployment, however it uses its own API – implemented for a range of popular languages – for service interaction, rather than the standardised (REST) mechanisms that we adopt, and (ii) Openshift [17], which aims to provide a platform for running web applications using cloud resources. It too needs quite sophisticated skills to write applications in supported languages by using the command-line administration tools specifically designed for this platform.

The Generic Worker framework [18] has similar goals to our framework: it provides PaaS service based on Microsoft's Azure Cloud platform. 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.

Additionally, toolkits such as Soaplab[19], Opal[20] 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. These too differ from our framework in several ways:

1. We adopt a cloud infrastructure 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 services will be composed, consequently a data staging mechanism is provided to assist in the effective composition of services. The above tools do not consider data communication as part of 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 avoids the need to learn and use the description languages adopted in these tools ("Ajax Command Definition" in Soaplab, "serviceMap" in GFac and "Metadata" in Opal), as well as the overheads involved in authoring, debugging and maintaining such descriptions in parallel with the application.

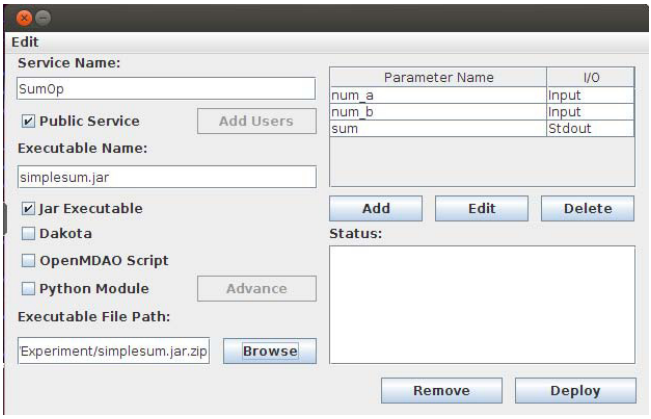
Our framework should be deployable in any private cloud or any popular public cloud based as it is on a set of open-source tools and standard protocols. The data can also reside in any form of cloud computing storage, such as Dropbox, Ubuntu one, OwnCloud or SpiderOak, for example. We also note that 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 have developed 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 Cloud-Based Framework for Scientific Applications

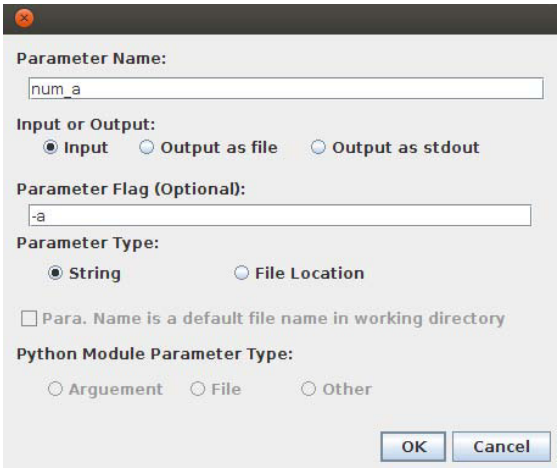
In this section, we describe our framework and how we believe it addresses the issues raised by the hurdles we identified earlier. We approach these issues from three perspectives: (i) service deployment, (ii) service invocation and execution, (iii) data staging.

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



(a) The main window of GUI tool



(b) The parameter window of GUI tool

Fig. 1. Windows of GUI tool

stage: (i) to upload and store the application and its dependencies in the cloud repository, (ii) to write and upload the description of the application to cloud for subsequent configuration and deployment, (iii) the configuration of authorization information that controls who may access the service once deployed. These tasks are all performed through the client GUI tool.

To illustrate the features of the deployment service, we use the screenshots shown in Figure 1, where Figure 1(a) shows the main window of the GUI tool. Our aim here is to make deployment tasks fit within the familiar range of operations of a desktop window manager. The GUI tool is set up to connect with the delpoyment service in the cloud through a URI with user authentication information. For the application uploading task, the user packs the binary and dependencies into a self-contained folder as a compressed

Fig. 2. Local folder for service description

Table 1. URIs of Datapool and Application Services

	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}

file and uploads it cloud side through the deployment service. The uploader can be started from the menu when the user right-clicks on the compressed file[2]. 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". One another notable feature shown in Figure 1(a) is the access permission setting. The user can choose whether a service can be accessed by all users as a *public service* or by selected users. Permitted users can be added in a separate window by the service owner clicking the *Add Users* button. Figure 1(b) shows the parameter window of the GUI tool. In the deployment process of Web service, the framework needs the information for mapping each command-line argument into a parameter for the web service. At the same, the framework also needs to generate a command-line for the invocation of the program. Therefore, this window allows the description of a wide range of command-line I/O types, such as argument flag, file path, standard I/O stream, etc. The framework identifies the binary file type through the extension name of file name entered here as well.

[2] 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.

Lastly, users also need functions to remove, modify or redeploy the service, which requires the service description. During the deployment process, the description – represented as a XML file – is uploaded as a cloud resource. At the same time, a copy is stored in a designated local folder. 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

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

Of particular note are the datapool resources: each denotes a collection of data items, addressable through an unique URI. Multiple Datapool instances can be generated and customized through the Datapool service by the user. Each data item inside a Datapool is also given an unique URI. Only the creator of each Datapool and the creator's services can access the content, which is ensured by the role-based authorization mechanism. There are two advantages to organizing data in this way. First, because all the data items and the data collection are directly associated with URIs, they are all web resources that can be accessed over 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 Table 1 illustrates the unified format.

Figure 3 shows an example deployment using the framework. It contains one client and two servers. Each server is composed of a pair of a Datapool and an Application service, both of whose implementation is based on Apache-Tomcat. All the components communicate with each other through REST services invocations. The execution of application service depends on the data provided by its local Datapool, which are fed through a file system. Figure 4 shows more details about the execution sequence in an example workflow based on the framework in Figure 3. In this example, Application

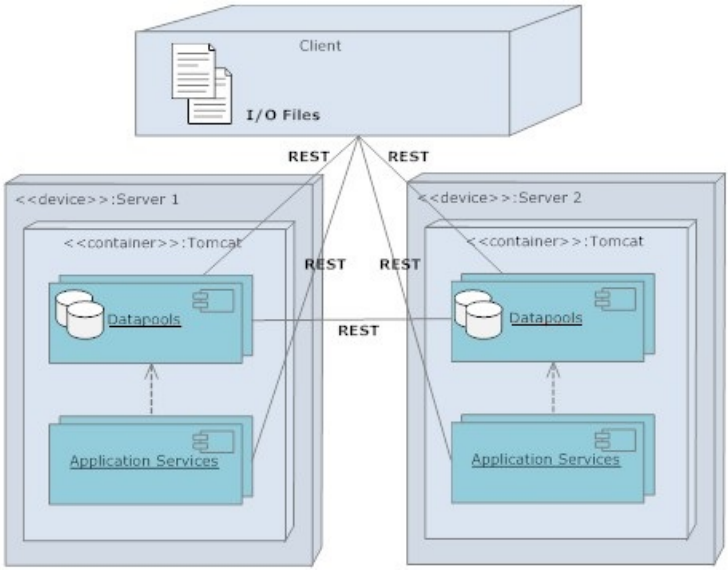


Fig. 3. The UML Deployment Diagram of the Framework Deployment Example

Service 1a(AS1a) consumes input D1. AS2a needs D2 and D3, which is the output generated by AS1a, as inputs. As depicted, client's duties are simplified to initializing input data and dispatching control signals to Datapool and Application Services. There are two essential features, which we emphasize here, namely: (i) inputs are uploaded to Datapool separately and in advance, so that Step 1 and Step 2 are able to execute concurrently (ii) DP2a can retrieve the input directly for AS2a in Step 12 and 13 from the other Datapool service without data needing to pass via the client.

3.3 The Data Staging Mechanism

Data staging and how to control it are not new problems. Already in 1997 [21], adopted the idea of distributed data-flows in a service composition framework to improve data transfer performance, as did also [22] some years later. Similar ideas are embodied in some distributed program execution engines, such as [23, 24], to overcome the bottleneck of data transfers. Meanwhile, several workflow management systems took up a peer-to-peer style mechanism for intermediate data movement[25–27]. 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 addressability: 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.

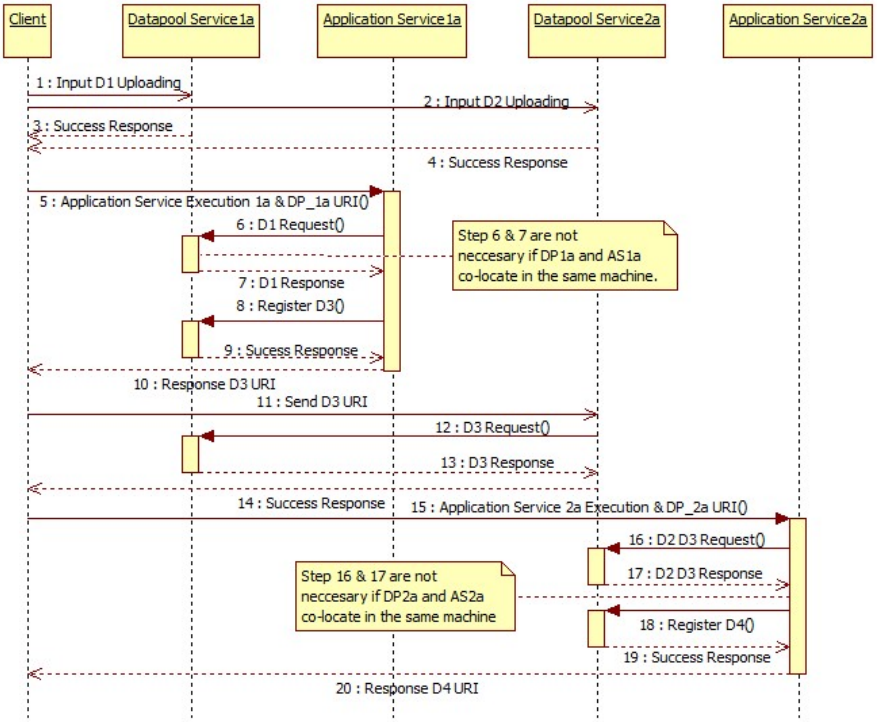


Fig. 4. The UML Sequence Diagram of the Execution of Workflow Example

We can make two quite obvious remarks about dataflows between several services: (i) for a given service invocation, the dataflow rarely involves the client or central controller, which means that dataflows can (normally) be distributed (point-to-point), and (ii) it is not uncommon that the necessary data objects (inputs) may come from different sources, suggesting 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.

Distributed Data Transfer. Figures 5 and 6 illustrate the essential difference between a centralized and a distributed mechanism for data transfer. Figure 5 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 6, 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

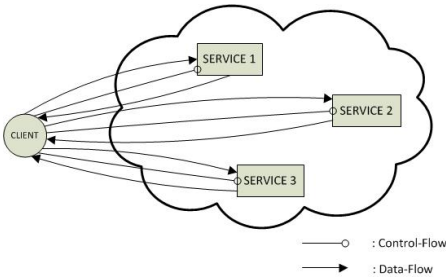


Fig. 5. Centralized Data-Flows in Web Services Compostion

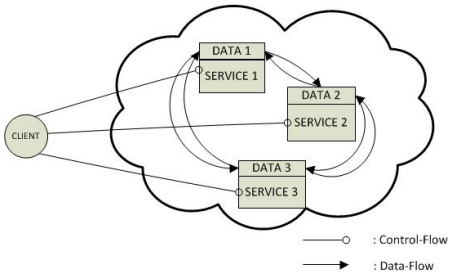


Fig. 6. Distributed Data-Flows in Web Services Composition

object has the capability to be identified and accessed universally through the Internet by means of its URI.

Asynchronous Data Transfer. 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, with an asynchronous method, the transfers start as and when each preceding service finishes. The transfers are not synchronized with the invocation of the next service, rather data elements are transferred and stored in the 'next' Datapool in advance, the benefits of which are analysed in [28].

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 the evaluators. 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 time to complete the operation 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

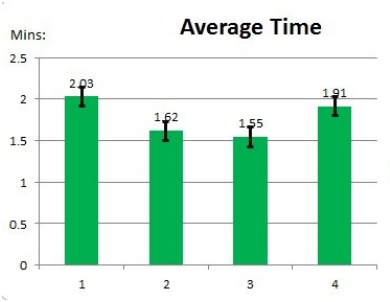


Fig. 7. The average time of deployment operations

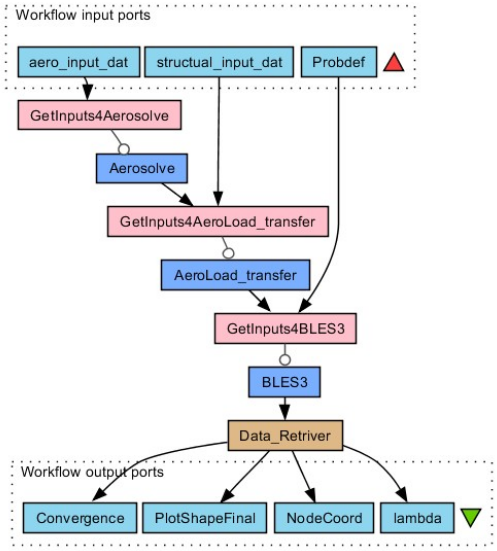


Fig. 8. The wing structure optimization process built in Taverna

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 for single service deployment.

4.2 Case Studies

Image Processing Workflow. In this workflow, the binaries for PovRay[29] and ImageMagick[30] are installed on the cloud-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 using ImageMagick. Both of their execution processes are written as Unix shell scripts. The uploaded package also includes related PovRay include files that serve as libraries for 3-D image generation. They are all deployed through the GUI tool as web services. In the deployment process, PovRay dependency files in the format of `inc` are 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 4).

This case study serves to demonstrate how the binary versions of two command-line programs with libraries 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 [11] to carry out the tasks of composition, execution and monitoring, as in our previous work [28, 31]. The composition of services expressed as a workflow, is also able to operate in conjunction with the distributed data staging mechanism of our framework, even though the intermediate data movement in Taverna is centralized in style. Figure 8 shows a screenshot of the service composition design example, which serves to optimize the internal stiffness distribution of a typical aircraft wing under coupled aerodynamics and structural considerations. In Figure 8, the boxes *Aerosolve, AeroLoad_transfer, BLES3* are services deployed based on three command line programs, written in Fortran and C. The boxes *GetInputs4Aerosolve, GetInputs4AeroLoad_transfer, GetInputsBLES3* are the Datapool services. The input ports built into Taverna are located at the top of Figure 8, and the output ports are at the bottom. One local service, *Data_Retriever*, 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[9] and Dakota[10]. OpenMDAO is based on Python and a workflow is expressed as an executable python script. With the support of the OpenMDAO runtime installed in a server (ie. cloud side), the deployment process can be achieved as easily as for any other command-line program. Dakota has a different execution approach in that the workflow is defined as a input file, which is then 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.

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 sharing of the discipline knowledge that is embedded in software.

4.3 Comparison of Data Staging Performance

In order to evaluate the performance of services deployed using our new framework, we have run the wing optimization process from Section 4.2 in two network-based configurations: (i) with all the programs deployed as SOAP services and controlled through a centralized client, including all the data transfers, constituting in effect a worst case scenario for data overheads, and (ii) with the programs deployed as REST services, using a centralized client for control, but the universal distributed flows framework for

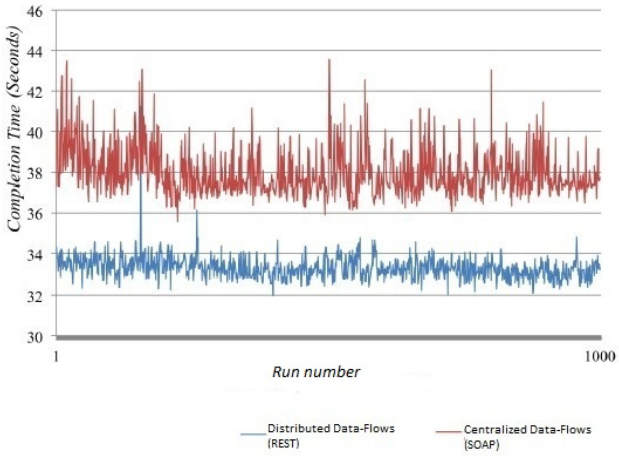


Fig. 9. Comparison of 1000 continuous executions

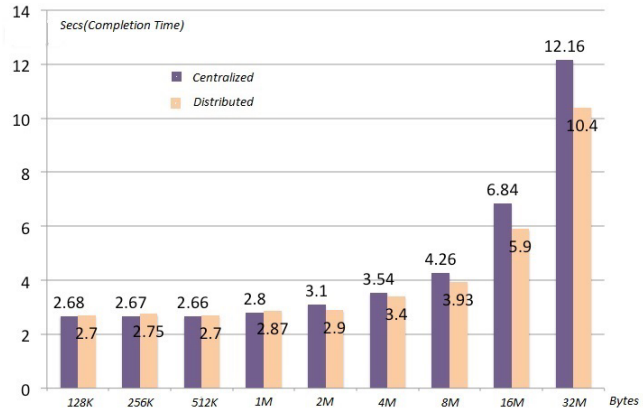


Fig. 10. Results of simple workflows with centralized and distributed data-flows

data. We first compare these two modes, where the programs or services are executed in the same machine environment and the network environment is also the same.

To provide preliminary evidence that the REST web services with distributed data-flows performs better than the centralized approach, we ran an experiment of 1000 consecutive executions for both processes in the same environment. The result is presented in Figure 9[3]. We can observe some spikes because of a changing network situation, but the figure shows that the REST workflow is faster by a clear margin and also demon-

[3] The x-axis only denotes the number of the run: it does not signify concurrent execution of the two modes. The data from the two sets of runs is overlaid to facilitate comparison of the execution times.

strates lower variation. In order to asses data transfer costs, we wrote a workflow that just moves data from client to one service, on to another, then back to the client. These two services are deployed in two different VMs on the same LAN as the client. Client and servers access each other by URIs. We set up two scenarios both using RESTful services, but while one uses centralized transfer, the other uses the distributed method. In the first scenario, the data transferred from the first service to the second is included in the HTTP body, while in the second just the URIs are transferred and data is transferred in the background by the Datapool service. The results are shown in Figure 10. Each workflow was run 10 times for the two scenarios and different data sizes to obtain the mean value. The results suggest the expected trend, in that gains increase with the size of data to be transferred. Crossover, in the test environment, occurs between and 1 and 2Mb, but clearly this will be different for different network environments.

5 Conclusion and Future Work

In this paper, we have presented evidence for the benefits arising from our light-weight 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. The framework reduces the effort for users to turn legacy codes and programs into web services and hence collaborate with each other. 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 [25]. A more serious issue however, is the dependence on specific 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 [32], 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 own services, and take advantage of the power offered by service composition to enable collaboration. 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 [33].

Acknowledgements. We thank Lizzie Gabe-Thomas for advice on experiment design in user trials of the deployment tools and the participants for their help.

References

1. Gannon, D., Ananthakrishnan, R., Krishnan, S., Govindaraju, M., Ramakrishnan, L., Slominski, A.: Grid Web Services and Application Factories. In: Grid Web Services and Application Factories, pp. 251–264. John Wiley & Sons, Ltd. (2003)
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), 249–260 (2006)
3. Sneed, H.M.: Integrating legacy software into a service oriented architecture. In: Proceedings of the 10th European Conference on Software Maintenance and Reengineering, CSMR 2006, p. 11. IEEE, Bari (2006)
4. Gorder, P.F.: Coming soon: Research in a cloud. Computing in Science and Engineering 10(6), 6–10 (2008)
5. Sullivan, F.: Guest editors introduction: Cloud computing for the sciences. Computing in Science & Engineering 11, 10 (2009)
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), 34–43 (2010)
7. Mell, P., Grance, T.: The nist definition of cloud computing (draft). NIST special publication 800(145), 7 (2011)
8. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. PhD thesis, University of California, Irvine (2000)
9. NASA Glenn Research Center: OpenMDAO, `http://openmdao.org/` (accessed January 15, 2014)
10. Sandia National Laboratories: The DAKOTA Project, `http://dakota.sandia.gov/` (accessed January 15, 2014)
11. School of Computer Science, University of Manchester: Taverna, `http://www.taverna.org.uk/` (accessed January 15, 2014)
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), 11–14 (1996)
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), 243–276 (2005)
14. Google: Google App Engine, `http://developers.google.com/appengine/` (accessed January 15, 2014)
15. Lindenbaum, J., Wiggins, A., Henry, O.: Heroku (2008),`http://www.heroku.com` (accessed January 15, 2014)
16. GoPivotal, Inc.: Cloud Foundry, `http://www.cloudfoundry.com/` (accessed Augest 24, 2014)
17. Red Hat, Inc.: Openshift, `https://www.openshift.com/` (accessed January 15, 2014)
18. Simmhan, Y., van Ingen, C., Subramanian, G., Li, J.: Bridging the gap between desktop and the cloud for escience applications. In: IEEE 3rd International Conference on Cloud Computing (CLOUD), pp. 474–481. IEEE, Chengdu (2010)
19. Senger, M., Rice, P., Bleasby, A., Oinn, T., Uludag, M.: Soaplab2: more reliable Sesame door to bioinformatics programs (2008)
20. Krishnan, S., Clementi, L., Ren, J., Papadopoulos, P., Li, W.: Design and evaluation of opal2: A toolkit for scientific software as a service. In: 2009 World Conference on Services - I, pp. 709–716. IEEE, Los Angeles (2009)
21. Alonso, G., Reinwald, B., Mohan, C.: Distributed data management in workflow environments. In: Proceedings of the Seventh International Workshop on Research Issues in Data Engineering, pp. 82–90 (April 1997)

22. 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)
23. 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 2011, p. 9. USENIX Association, Berkeley (2011)
24. 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 2007, pp. 59–72. ACM, New York (2007)
25. Davis, U.C., Santa Barbara, U.C., San Diego, U.C.: Kepler project, `https://kepler-project.org/` (accessed: January 15, 2014)
26. Cardiff University: Triana project, `http://www.trianacode.org/` (accessed May 08, 2013)
27. Cao, J., Jarvis, S., Saini, S., Nudd, G.: Gridflow: workflow management for grid computing. In: Proceedings of the CCGrid 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid, 2003, pp. 198–205 (May 2003)
28. 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 Third International Workshop on RESTful Design, pp. 41–48. ACM, Lyon (2012)
29. Persistence of Vision Raytracer Pty. Ltd.: Povray, `http://www.povray.org/` (accessed Januray 15, 2013)
30. ImageMagick Studio: Imagemagick, `http://www.imagemagick.org` (accessed January 15, 2014)
31. Duan, K., Seowy, Y.V., Kim, H.A., Padget, J.: A Resource-Oriented Architecture for MDO Framework. In: Proceeding of 8th AIAA Multidisciplinary Design Optimization Specialist Conference, AIAA, Honolulu (2012)
32. 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, pp. 371–378. IEEE, Bangalore (2007), doi:10.1109/E-SCIENCE.2007.48
33. Fukuta, N., Satoh, K., Yamaguchi, T.: Towards "Kiga-kiku" services on speculative computation. In: Yamaguchi, T. (ed.) PAKM 2008. LNCS (LNAI), vol. 5345, pp. 256–267. Springer, Heidelberg (2008)