# Architectures for workflow integration in science gateways

**Abstract**

*Keywords:*

# Architectures for workflow integration in science gateways

## 1. Introduction

[*Say that the review is done based on our experience with VIP, CBRAIN, and to some extent SHIWA.*]

*Context.* The question of integrating workflow engines in science gateways can be seen at various levels, corresponding to various definitions of workflows. One level is the SHIWA level, where it was considered that workflow engines are aware of the DCI (and 'D' is important because of data transfers, proxies, etc). Another level is to remove 'D' from the definition: a workflow engine becomes a program that submits jobs, potentially only to local clusters. It opens a whole new class of workflow engines that we used to consider as "applications". For instance, in neuroinformatics: Nipype, PSOM, but also FSL through the fslsub tool. A workflow engine is not supposed to be aware of the science gateway. A wide-array of workflow engines are available with specificies coming from the application domain, available tools, etc Their integration in science gateways becomes critical. Several of the examples presented in the paper will be taken from medical image analysis, in particular neuroimaging.

*Goal.* this paper reviews and compares the architectures to integrate workflow engines in science gateways.

*Contributions.*

- We evaluate architectures based on our experience with existing systems
- We propose a new architecture

## 2. Workflow engines and science gateways

### *2.1. Workflow engines*

In the last decade, the e-Science workflow community has developed high-level workflow systems to help developers access distributed infrastructures such as clusters, grids, clouds and web services, resulting in tools such as Askalon [1], Hyperflow [2], MOTEUR [3], Pegasus [4, 5], Swift [6], Taverna [7], Triana [8], VisTrails [9], WS-PGRADE/gUSE [10], etc. Such workflow engines usually describe applications in a specific language that offers parallel data and control flow constructs, visual edition features, links with domain-specific tool repositories, provenance recording, etc.

At the same time, specific toolboxes were emerging in various scientific domains to facilitate the assembly of software components in consistent pipelines. For instance, in neuroimaging, our primary domain of interest, tools such as Nipype [11], PSOM [12], SPM [13] and FSL [14] provide abstractions and functions to handle the data and computing flow between processes implemented in a variety of programming languages. Such tools were interfaced to computing systems, in particular clusters, to create tasks, handle their dependencies, and execute them. For instance, FSL can launch tasks on SGE through its `fsl_sub` tool, Nipype has execution plugins for S/OGE, Torque, IPython, ssh and multi-processor servers, and PSOM [...] . Some of these tools also support advanced features such as provenance tracking (Nipype, PSOM), redundancy detection across analyses to avoid re-computation (PSOM), and so on. A wide-array of workflows have been implemented using these pipelines and are now widely shared across neuroimaging groups world-wide. [*Provide more context on that?*] These domain-specific engines represent a tremendous opportunity for science gateways to leverage existing tools and applications. They nicely complement e-Science engines that are more oriented towards the exploitation of distributed computing infrastructures, in particular grids and clouds.

We are looking for software architectures that are able to integrate e-Science and domain-specific workflow engines in science gateways. For the sake of the analysis conducted in this paper, we define a *workflow engine* (also abbreviated *engine*)

as a software that submits interdependent computing *tasks* to an infrastructure (local server, cluster, grid or cloud) based on a workflow description (a.k.a *workflow*) and input data that may be files, data base entries, or simple parameter values. Although simplistic, this definition covers both e-Science and domain-specific engines. Some workflow engines, usually e-Science ones, may transfer data across the infrastructure, and others, usually domain-specific ones, may leave it to external processes. Workflows may be expressed in any language, including high-level XML or JSON dialects such as `Scufl` or `Hyperflow`, and low-level scripts such as `bash`. [*Mention other workflow definitions, e.g. abstract/concrete, etc?*]

### *2.2. Science gateways*

Science Gateway: An ecosystem, usually accessible through a web portal, that provides tools to access distributed infrastructures. Tools usually help users manage data transfers, task execution and authentication on multiple computing and storage locations. Examples: CBRAIN, VIP, NSG, etc

Multi-engine, multi-language.

Several workflow engines may be combined in the same science gateway.

A workflow engine is not supposed to be aware of the science gateway.

### *2.3. Infrastructures*

The infrastructure consists of the computing and storage resources involved in the workflow execution, and the software services used to access these resources. Infrastructures can be servers, clusters, grids or clouds. Some workflow engines may assume specific characteristics about the infrastructure, such as the presence of a shared file system between the computing nodes.

## 3. Architectures

The architectures are diagramed in Figure 2 using the graphical notations shown in Figure 1. Architectures are described from their main software *components* and *interactions*. Software components include science gateway and infrastructure in all architectures while workflow service, workflow pool and agent are involved in some architectures. The workflow engine itself is represented by a specific symbol with dotted lines. Software interactions

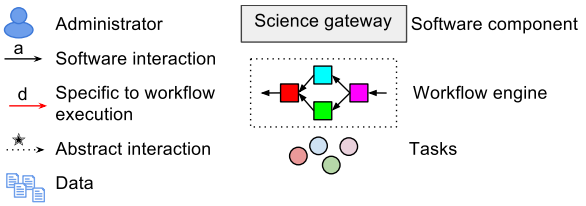



Figure 1: Graphical notations

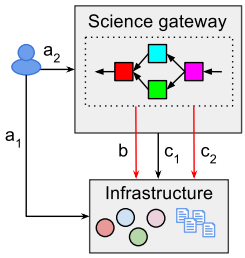
among these components are represented with arrows. *Abstract* interactions are specific types of interactions that may be implemented by various different software interactions. They are represented with dotted arrows and labeled with `*`. An architecture that has an abstract interaction is an abstract architecture. Red color is used to represent the components and interactions that are specific to workflow execution, i.e. that are not involved in any other process. Tasks refer to computing tasks that are created by the workflow engine and executed on the infrastructure. Data represents any type of file, or database that is involved in the workflow execution and stored on the infrastructure while the workflow executes. It does not cover workflow parameters such as strings, numbers, etc. The remainder of this section describes the different types of interactions involved in the architectures, and details each architecture.

### *3.1. Interactions*

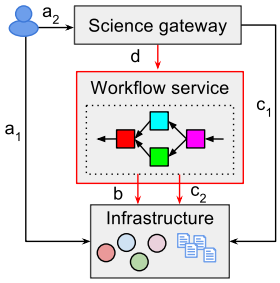
The interactions involved in the architectures are described below and labeled consistently with the notations used on Figure 2.

`(a)` Workflow integration: consists in adding a new workflow to the system so that users can execute it. It is triggered by an administrator of the science gateway and it results in an interface, for instance a web form, where users can enter the parameters of the workflow to be executed. The interaction has two aspects: ($\mathtt{a}_1$) the programs used in the workflow are installed on the infrastructure, which may or may not require administrative privileges on the infrastructure; ($\mathtt{a}_2$) the workflow is configured in the science gateway so that it becomes available to users. Note that integrating a workflow is not the same process as integrating a workflow *engine*.

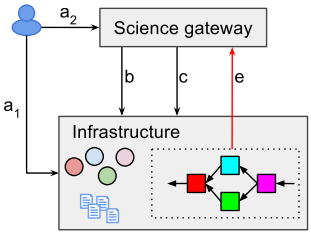
`(b)` Task control: operations to manage tasks on the infrastructure, including: authentication, submission, monitoring, termination, deletion, etc.



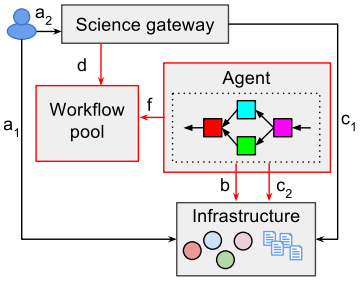

(a) Tight integration



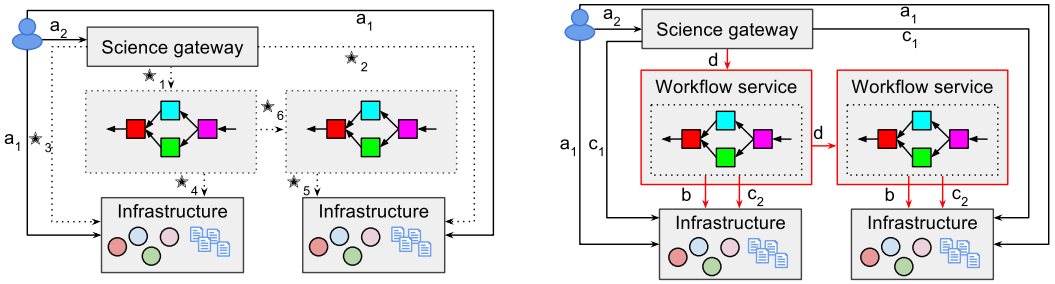

(b) Service invocation



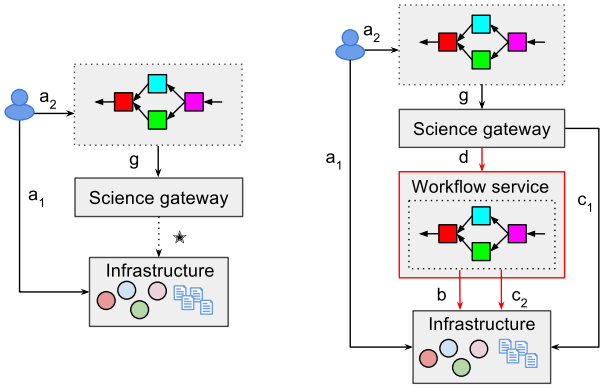

(c) Sub-tasking




(d) Pool




(e) Nested workflows. Left: abstract model. Right: instantiation with service invocation.




(f) Workflow import. Left: abstract model. Right: instantiation with service invocation.

Figure 2: Architectures

Controlling tasks requires to deal with the heterogeneous batch managers and meta-schedulers that might be available on the infrastructure. When the infrastructure is a grid or a cloud, it is for instance implemented using libraries such as SAGA, DRMAA, OCCI and similar initiatives *[refs needed]* .

(c) Data control: operations to manage data on the infrastructure, such as: upload, download, deletion, browsing, replication, caching, etc. Data movements can be triggered by the user in the science gateway ($c_1$), to upload input data or download processed data. They can also be performed by the workflow engine ($c_2$), to transfer data across the infrastructure. The infrastructure might offer various data storage backends with heterogeneous interfaces. Tools and services such as JSAGA [15] or Data Avenue [16] can be used to homogeneize these interfaces.

(d) Workflow control: operations to execute a workflow with an engine, including: workflow submission, monitoring, termination, etc. Workflow control can be coarse-grained (a.k.a. black box) or fine-grained (white box). In a coarse-grained model, the various tasks created by a workflow execution are masked and the user only has a global view of the workflow execution. In a fine-grained model, user is exposed to the workflow topology, i.e. to the outputs of the individual tasks, their statuses and so on.

(e) Sub-task control: operations used by tasks to submit sub-tasks on the infrastructure, including: submission, monitoring, termination, deletion, etc. Sub-task control is similar to interaction b, except that information about the parent task which submitted a sub-task is usually available and used for additional control. For instance, the parent task may wait for all its sub-tasks to finish before finishing, and conversely all the sub-tasks may be killed if the parent task is killed.

(f) Pool-agent: specific to the architecture described in Section 3.5. This is an interaction used when agents retrieve work from a central pool. It covers agent registration and de-registration to the pool, protocols to send work from the pool to the agent, mechanisms to update work status, and so on.

(g) Workflow conversion: translation from one workflow language to another one. This interaction may not be available or implementable for every workflow language. It has been developed mostly for well-structured and relatively simple workflow language such as between GWorkflowDL and Scufl [17] and between the 4 languages that were involved in the SHIWA initiative. In SHIWA, workflow conversion is done through an intermediate representation, the Interoperable Workflow Intermediate Representation [18], which allows to convert among $n$ workflow languages using $2n$ interactions instead of $n^2$.

### *3.2. Tight integration*

See Figure 2a. The workflow engine is tightly integrated with the science gateway, which means that it is deployed on the same machine and potentially shares code, libraries and other software components with the science gateway. For instance, the workflow engine might be a portlet in a Liferay portal or a controller in a Ruby on Rails application. The workflow engine and the science gateway usually share a database where application, users and other resources are stored. In this model, task and data control are both initiated from the science gateway. Interactions b and $c_2$ are initiated from the workflow engine while $c_1$ comes from other parts of the science gateway, for instance a data management interface. As in any other model, the installation of new workflows in the science gateway ($a_2$) and infrastructure ($a_1$) is done by an administrator, for security reasons. This is the model adopted in the Catania Science Gateway Framework [19] (see specific documentation on workflows[1]), CIPRES [20] and LONI Pipeline Environment [21]. *[Add description of a real system.]*

### *3.3. Service invocation*

See Figure 2b. The workflow engine is available externally to the science gateway, in a service. The science gateway controls the service through a specific interaction (d) that might be implemented as a web-service call (e.g. RESTful or SOAP), as a command-line or as any other method that offers a well-defined interface to the workflow engine. The workflow engine might be invoked either as a black box that completely masks the infrastructure and

[1] http://bit.ly/1oQrzvQ

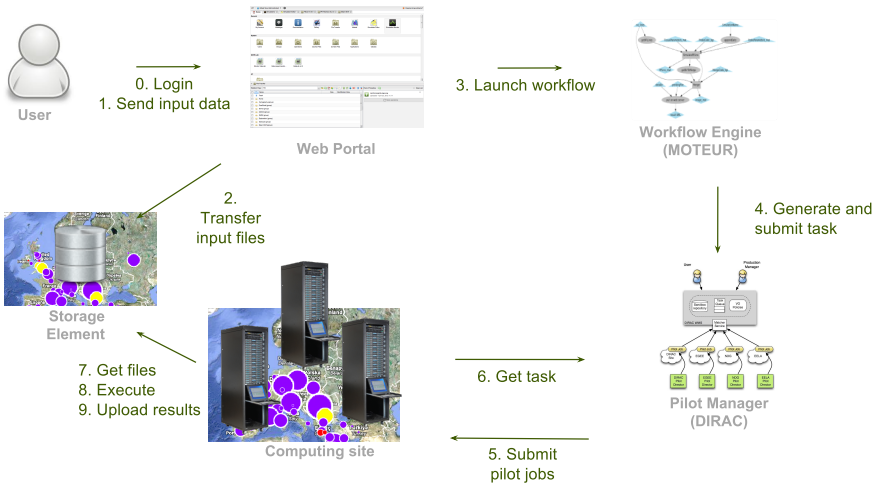



Figure 3: Architecture of the Virtual Imaging Platform that integrates the MOTEUR engine through service invocation.

tasks, or as a white box that allows for some interaction with them. The workflow engine is responsible for controlling the tasks on the infrastructure (`b`) and for performing the required data transfers to execute them ($c_2$). User data is usually managed through the science gateway ($c_1$), although it might as well be delivered by the workflow engine directly to the user.

This architecture is largely adopted, in systems such as Apache Airvata [22], Vine Toolkit [23], Virtual Imaging Platform [24], the WS-PGRADE/gUSE framework [10] and the numerous science gateway instances that rely on WS-PGRADE/gUSE [25]. Figure 3 shows the architecture of the Virtual Imaging Platform, where the MOTEUR workflow engine [3] is invoked as a service at step `3` (interaction `d` on Figure 2). Step `2` on Figure 3 corresponds to interaction $c_1$, and step `4` maps to interactions `b` and $c_2$ (in VIP, workflow data transfers are performed by the tasks and embedded in their descriptions).

### *3.4. Sub-tasking*

See Figure 2c. The workflow engine is a particular task that can submit sub-tasks to the science gateway through interaction `e`. The workflow engine keeps track of the dependencies between the sub-tasks but their execution is delegated to the science gateway that executes them on the infrastructure through interaction `b`. Although the science gateway has no global vision of the workflow, it can keep track of the sub-tasks submitted by a given task, for instance to be able to cancel them when the task is canceled. The science gateway may also implement mechanisms to facilitate the handling of task dependencies, for instance basic dependency lists as available in most batch managers (see for instance attribute `depend` of option `-W` in Torque[2]).

The science gateway also transfers both user and workflow data across the infrastructure, so that interactions $c_1$ and $c_2$ are both covered by `c`. In practice, both $c_1$ and $c_2$ can be implemented using the same pieces of code.

The sub-tasking architecture is implemented in CBRAIN [26] where it is used to integrate the PSOM workflow engine [12] and the FSL toolkit. The CBRAIN-PSOM integration [27] is illustrated on Figure 4. On this Figure, the science gateway is represented by components `CBRAIN portal` and `CBRAIN execution server` and the infrastructure consists of `Storage servers` and `Computing cluster with shared file system`. Interaction `b` on Figure 2 is implemented by steps `4`, `5`, `7` and `8` (regular interactions with batch manager). Interaction `c` is implemented by steps `3` and `10`. Interaction `e` is implemented by step `6`. The PSOM workflow engine adopts a pilot-job architecture [28] where the a master coordinates workflow execution by submitting workers and establishing direct communication channels with them. Note how this peculiar execution model is totally supported by the sub-tasking architecture.

The CBRAIN-FSL integration is also remarkable as it allows to leverage FSL pipelines that are writ-

[2] `http://docs.adaptivecomputing.com`

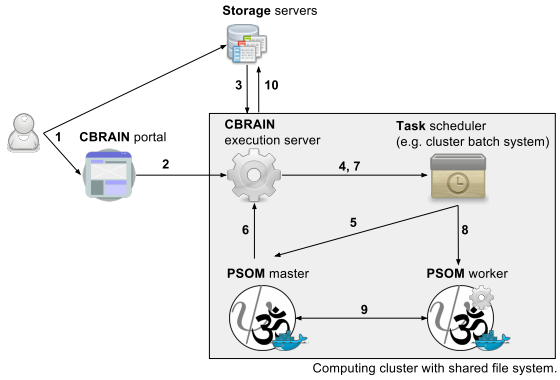



Figure 4: Architecture of the CBRAIN-PSOM integration that illustrates the sub-tasking model (Figure extracted from [27]). 1: User sends data and workflow execution request to storage server(s) and CBRAIN portal. 2: CBRAIN portal sends execution request to execution server on cluster. 3: Execution server transfers data from storage server(s). 4-5: Execution server starts PSOM master via task scheduler. 6: PSOM master submits PSOM workers to execution server. 7-8: Execution server starts PSOM workers through task scheduler. 9: PSOM master and workers execute workflow. 10. Execution server transfers results to storage server(s). [*Improve graphics*]

ten in a very low-level workflow language (Linux executables and script) that submits tasks uniformly through a specific tool called `fsl_sub`. To integrate FSL in CBRAIN, `fsl_sub` was modified to implement interaction $\mathtt{e}^3$ [*Add a diagram and maybe a code excerpt to illustrate that.*] .

### *3.5. Pool model*

See Figure 2d. Workflows are submitted by the science gateway to a pool through interaction `d`. Agents connect to the pool asynchronously to retrieve and execute workflows through interaction `f`. Agents may be started according to various policies, for instance to ensure load balancing. Agents may wrap different types of workflow engines. Workflow engine controls tasks and data on the infrastructure through interactions `b` and $\mathtt{c}_2$, science gateway transfers user data through interaction $\mathtt{c}_1$, and administrator installs workflows through interaction $\mathtt{a}_1$ and $\mathtt{a}_2$.

The pool model was implemented in the SHIWA pool [29], see Figure 5. In Figure 5, interaction `d` of Figure 2d is implemented in 3 different types of calls to the pool: workflow submission (`1` & `2`), workflow status retrieval (`a` and `b`), and workflow retrieval (`13` & `14`). In Figure 5, interaction `f` of Figure 2d is implemented through 2 types of calls: workflow instance retrieval (`4`, `5` and `6`), and workflow instance update (`11` and `12`). Workflow instance retrieval is used by the agents to fetch work from the pool. Workflow instance update is used by the agents to update workflow statuses. In the SHIWA pool, agents can wrap different types of workflow engines to execute workflows expressed with different languages. Calls `0`, `7`, `8` and `9` on Figure 5 are used by workflow engine plugins to declare their supported language and launch engines, and by workflow engines to report their status to the agent. These calls are specific to the SHIWA Pool implementation of the `agent` component and therefore have no corresponding representation in Figure 2d.

### *3.6. Nested workflows*

See Figure 2e. In nested workflows (Figure 2e-Left), a task of a *parent* workflow executed by the *parent* engine is itself a workflow, called *child* workflow, that is executed by a *child* engine. The parent and child engines might use different workflow description languages. They might also execute workflows on different infrastructures. The parent workflow is also called meta-workflow. The science gateway communicates with the parent engine through interaction $*_1$. The science gateway also communicate with the infrastructure to transfer user data through abstract interactions $*_2$ and $*_3$. Both workflow engines communicate with the infrastructure through abstract interactions as well, $*_4$ and $*_5$. The parent engine communicates with the child engine through abstract interaction $*_6$. Administrator installs workflows in the science gateway through interaction $\mathtt{a}_2$, and installs software on infrastructures through interactions $\mathtt{a}_1$.

Nested workflows are abstract architectural patterns that can be instantiated in the various architectures described previously. We focus on instantiation with the service invocation model (see Figure 2e-Right) as this is the most used architecture. In such an instantiation, we assume that the parent and child workflow engines are distinct pieces of software that require different workflow services invoked by distinct `d` interactions. If this is not the case, then workflow services can be collapsed in a single one with a `d` interaction with itself. Workflow engines communicate with infrastructures us-

[3] See `fsl_sub` modified script available at `https://github.com/aces/cbrain-plugins-neuro`

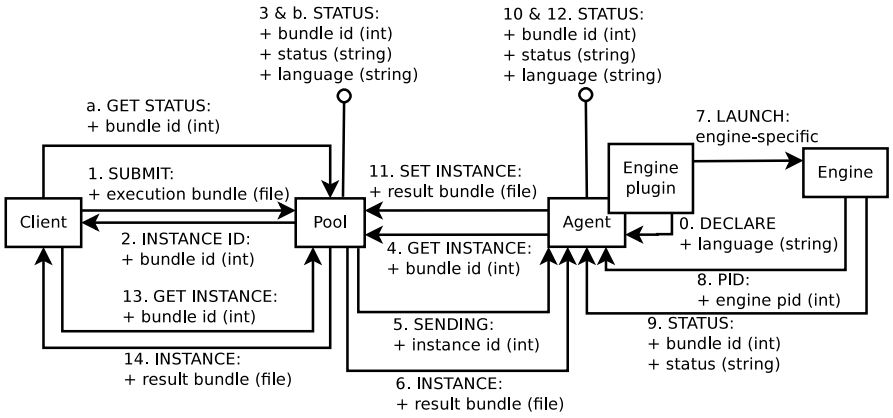



Figure 5: Architecture of the SHIWA pool (Figure reproduced from [29]). Circle-terminated arrows indicate messages that are broadcast to all pool clients.

ing `b` and $c_2$. Science gateway transfer user data to infrastructures using $c_1$ interactions.

Nested workflows have long been available in workflow engines, for instance in the Taverna workbench [7]. They are also used implicitly in several platforms where workflow engines are wrapped in workflow tasks as any other command-line tool. Nested workflows were notably used by the SHIWA Science Gateway to implement so-called Coarse-Grained workflow interoperability [30], i.e. to integrate various workflow engines in a consistent platform. Figure 6 shows the architecture used in the SHIWA Science Gateway for nested workflow execution with service invocation as represented in Figure 2e. The parent workflow engine is `WS-PGRADE`, invoked as a service in the Science Gateway (step `1` on Figure 6, interaction `d` on Figure 2e). Ten different child engines can be used by nested workflows, invoked through the `Submission service` (step `2`, interaction `d`). Each of these engines can submit jobs and transfer data to a distributed computing infrastructure (`DCI`, step `3`, interactions `b` and $c_2$). Data interactions (`c`) and application porting ones (`a`) are not represented on Figure 6.

### *3.7. Workflow import*

See Figure 2f. This is an abstract model that we instantiate with the service invocation architecture for consistency. Workflows are integrated in the science gateway through format conversion from a native format to the science gateway format. This was implemented in the SHIWA Science Gateway through the IWIR language that provided a common language for portability across grid workflow systems [31].

*[workflow import is an offline process.]* *[Add description of a real system]*

## 4. Evaluation

The architectures described in Section 3 are evaluated in Table 1. We use 5 main criteria to evaluate the architectures: integration effort, robustness of workflow execution, extensibility, scalability and other features. Criteria break down to specific metrics where *low value indicates good performance*. For each criterion, a total score is computed by summing up the individual metrics. We ensure that the different metrics in a criterion measure similar entities so that summation makes sense. The criteria and metrics are explained hereafter.

### *4.1. Evaluation metrics*

*Integration effort* measures the development required to build the architecture. It is obtained by counting the total number of interactions and components on the architecture diagram. It breaks down to the following 2 metrics:

- Total number of components ($I_1$),
- Total number of interactions ($I_2$).

One may wonder whether infrastructure should be counted in $I_1$ since it is usually not developed by the groups who integrate workflow engines in science gateway. However, integrating an infrastructure in the architecture usually requires some technical effort (e.g. account creation, software installation, etc) which is why we keep it in the metric. The two metrics are summed to obtain the total score

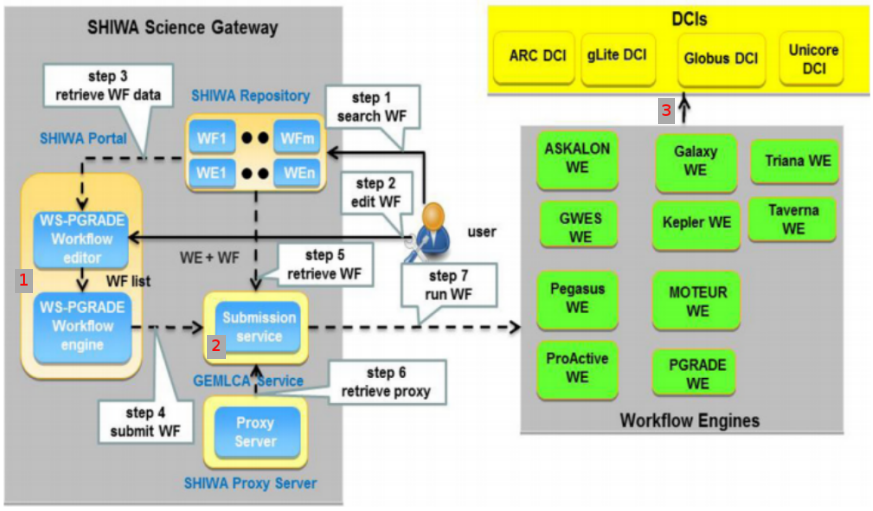



Figure 6: Nested workflow execution through SHIWA Science Gateway (Figure reproduced from [30] with permission of the first author).

for this criterion, which measures the total number of software pieces to develop.

*Robustness of workflow execution* measures the likelihood that workflow execution fails due to errors in the components or with the interactions in the software architecture. Robustness is determined as the number of software components and interactions that are specific to workflow execution, i.e. not reused by other functions:

- Components ($R_1$): number of specific components involved in workflow executions. Science gateway, for instance, is *not* specific to workflow execution since it is used to authenticate users, add new workflows, transfer user data, etc.
- Interactions ($R_2$): number of specific interactions involved in workflow executions.

These specific components and interactions are represented in red in the architecture diagrams in Figure 2. $R_1$ and $R_2$ assume that complex architectures tend to be prone to failure. $R_1$ and $R_2$ are summed to obtain the total score for this criterion, which measures the total number of software pieces that are specifically involved in workflow execution.

*Extensibility* measures the difficulty to replace or add elements in the architecture. It is determined as the number of interactions and components to modify when a new element is added. Modification of a component is required when its code needs to be modified or recompiled (science gateway or workflow service), or when a new piece of software has to be installed (infrastructure only). Modification of an interaction is deemed necessary when the parameters involved in this interaction are modified. Extensibility breaks down in 4 metrics depending on the type of element that has to be added or replaced:

- New engine type ($E_1$): number of interactions or components to modify to integrate a new type of workflow engine in the architecture. Workflow engines belong to different types when they cannot be invoked using the same interface. Adding a new type of workflow engine allows to execute more workflows in the science gateway.
- New engine version ($E_2$): number of interactions or components to modify to integrate a new version of a workflow engine in the architecture, assuming that another version of the same engine type is already available. Different versions of a workflow engine share the same interface, i.e. they can be invoked using the same software. When this is not the case, the different versions have to be considered as different engine types.
- New workflow ($E_3$): adding a new workflow is a very common operation that does not require modifying software components or interactions. We measure the difficulty to integrate a new workflow by counting the number of interactions that need to be invoked to integrate a new workflow in the architecture, assuming that the engine type and version required to execute this workflow are already avail-

able.

• New infrastructure ($E_4$): number of interactions or components to modify to integrate a new type of infrastructure in the architecture. Adding a new infrastructure allows to provide more computing or storage power, to access specific types of resources (e.g. GPUs, clouds), or to enforce execution policies (e.g. constrain data to remain in a particular network domain).

The 4 metrics are summed to obtain a global index that measures the difficulty to extend the architecture.

*Scalability* corresponds to the ability of the architecture to cope with high workloads. It is measured by counting the potential scalability issues in the architectures, i.e. the lack of scalability features. Features are evaluated using a 3-level metric: 0 means that the feature is very easy to enable, 1 means that it can be implemented but with some difficulty, and 2 means that the feature could only be implemented with a nonsensical amount of effort. Four different features are identified:

• Multiple engine instances ($S_1$): possibility to have more than 1 engine instance in the architecture. Workflow engines may require important amounts of resources when a lot of workflows, or large workflows, are executed. At some point, it may be required for the science gateway to distribute the load to several engines. $S_1$=0 when adding a new engine instance is a fully automated process, i.e. the workflow engine only has to be started. In this case, elastic engines are possible, i.e. some kind of auto-scaling mechanism can be implemented to control the number of engine instances in the architecture. $S_1$=1 when adding a new engine instance requires some form of manual intervention, which prevents easy implementations of elastic engines. $S_1$=2 when new engine instances cannot be added.

• Distributed engines ($S_2$): possibility to distribute the execution *of a single workflow* among different engine instances. In our scope, this feature focuses on the capabilities of the architecture rather than these of the workflow engine. $S_2$=0 when distributed engines are possible in the architecture and $S_2$=1 when they require specific developments in the workflow engine.

• Task scheduling ($S_3$): task scheduling is a difficult issue that depends more on the implementation of specific algorithms in the science gateway, workflow engine and infrastructure than on the architecture used to integrate the workflow engine in the science gateway. Some architectures, however, complexify the task scheduling problem by introducing additional software layers or create tasks with specific characteristics. $S_3$=0 when the architecture does not introduce any additional complexity to the scheduling problem and $S_3$=1 otherwise.

The 3 metrics are summed to obtain a global measure of the scalability issues that are present in the architecture.

*Other features* include:

• Meta-workflow ($O_1$): ability to describe meta-workflows from existing workflows. [*More details about why this is important?*] $O_1$=0 when the feature is available, $O_1$=1 otherwise.

• Fine-grained debugging ($O_2$): availability of fine-grained debugging information about workflow tasks. When such information is available, workflow is seen as a white box. Fine-grained information about workflow tasks is required to properly troubleshoot workflow executions. $O_2$=0 when such information is easy to access, $O_2$=1 otherwise. [*More details about why this is important?*]

$O_1$ and $O_2$ are summed to obtain a total number of missing other features in the architecture. [*Maybe we could revise the categories to avoid such a vague "other features".*]

The different architectures described in Figure 2 are evaluated along these metrics in the remainder of this Section.

### *4.2. Tight integration*

*Integration.* This architecture does not require any component in addition to the science gateway and infrastructure ($I_1$=2). It involves 5 interactions: $a_1$, $a_2$, b, $c_1$ and $c_2$ ($I_2$=5).

*Robustness.* No component is specific to workflow execution ($R_1$=0), but interactions b and $c_2$ are ($R_2$=2).

*Extensibility.* Integrating a new type of workflow engine requires to modify the science gateway as well as interactions b and $c_2$ ($E_1$=3). Updating a workflow engine version requires modifications in the science gateway ($E_2$=1). Inserting a new workflow is done through interactions $a_1$ and $a_2$ ($E_3$=2). Adding a new infrastructure generates updates in interactions b, $c_1$ and $c_2$ ($E_4$=3).

*Scalability.* Adding a new engine instance requires a new instance of the science gateway, which is in general not possible ($S_1$=2). Distributed engines are not available by default ($S_2$=1). The scheduling of tasks on the infrastructure is as complex as in any other architecture since the workflow engine might implement any kind of scheduling policy ($S_3$=0).

*Other features.* Meta-workflows are not supported by default ($O_1$=1). Debugging is not an issue since the science gateway can retrieve any information from the workflow engine directly ($O_2$=0).

### *4.3. Service invocation*

*Integration.* Service invocation requires a workflow service in addition to the science gateway and infrastructure ($I_1$=3). The architecture involves 6 interactions: $a_1$, $a_2$, b, $c_1$, $c_2$ and d ($I_2$=6).

*Robustness.* The workflow service is a component specific to workflow execution ($R_1$=1). Workflow execution also involves 3 specific interactions: b, $c_2$ and d ($R_2$=3).

*Extensibility.* Adding a new type of workflow engine requires to implement the corresponding workflow service, to modify interaction d, and to implement interactions b and $c_2$ ($E_1$=4). New engine versions can be added by updating the workflow service without modifying any interaction ($E_2$=0). New workflows are added in the science gateway or in the workflow engine through interactions $a_1$ and $a_2$ ($E_3$=2). Adding a new type of infrastructure requires updates in interactions b, $c_1$ and $c_2$ ($E_4$=3).

*Scalability.* The service architecture supports multiple engine instances through multiple workflow services. In VIP for instance, this feature has been available from release 1.17 (April 2016). A basic load-balancing mechanism is available that sends new workflow executions to the engine instance that has the least active executions. To avoid "black-hole" syndromes created by failing engine instances, engine instances are automatically disabled when workflows cannot be submitted to them. Adding a new engine instance, however, requires manual intervention to declare the new instance in the science gateway ($S_1$=1). Consequently, elastic engines are difficult to implement because they require a mechanism to update the science gateway configuration when a new engine instance is available. Distributing the execution of a single workflow in multiple engines is usually not possible unless the workflow engine has specific abilities ($S_2$=1). The scheduling of tasks on the infrastructure is as complex as in any other architecture since the workflow engine might implement any kind of scheduling policy ($S_3$=0).

*Other features.* Meta-workflow are not supported by default ($O_1$=1). Fine-grained debugging information is usually easy to obtain since the workflow service provides direct access to the engine ($O_2$=0).

### *4.4. Sub-task*

*Integration.* The sub-task architecture requires only 2 components ($I_1$=2). It involves 5 interactions: $a_1$, $a_2$, b, c and e ($I_2$=5).

*Robustness.* No component is specific to workflow execution ($R_1$=0), and only interaction e is ($R_2$=1).

*Extensibility.* Integrating a new type of workflow engine requires to develop interaction e and to install the engine on the infrastructure ($E_1$=2). Updating an engine version requires updates only on the infrastructure ($E_2$=1). New workflows are integrated by creating a new task in the science gateway through interactions $a_1$ and $a_2$ ($E_3$=2). Adding a new infrastructure requires to update interactions b and c in the science gateway ($E_4$=2).

*Other features.* Meta-workflows are not available ($O_1$=1). Obtaining fine-grained information about workflow tasks is not straightforward since the science gateway has no knowledge about the workflow topology, and the workflow engine is integrated as a task ($O_2$=1).

*Scalability.* New engine instances are spawned and executed on the infrastructure as any other task upon user submission ($S_1$=0). This is a major interest of the sub-task architecture. Distributed engines are not supported by default ($S_2$=1). Task scheduling is slightly more complex than in the other approaches due to the special role of the task that executes the workflow engine ($S_3$=1). Indeed, the reliability of this task is critical since all the sub-tasks in the workflow depend on it and, depending on the recovery capabilities of the workflow engine, may need to be resubmitted if the workflow task fails. The workflow task is also longer than all its sub-tasks, which increases its chances of failure. In addition, task parameters, for instance estimated walltime, are more difficult to estimate for

the workflow task than for sub-tasks which may generate issues such as selection of wrong batch queues on clusters. Finally, the interdependencies between the workflow task and its sub-tasks may create deadlocks when there is contention. Say for instance that only 1 computing resource is available for the science gateway and that the workflow task is running on it and submits sub-tasks, then the sub-tasks could only execute when the resource is available, which will never happen because the workflow task will not complete until the sub-tasks complete. This configuration can be generalized to an infrastructure with $n$ resources where $n$ workflows are submitted. In practice, however, the number of submitted workflows usually remains lower than the number of computing resources available on this infrastructure, which makes such deadlocks unlikely to happen.

### *4.5. Pool*

*Integration.* The pool model requires a workflow pool and an agent in addition to the science gateway and infrastructure ($I_1$=4). It involves 7 interactions: $a_1$, $a_2$, b, $c_1$, $c_2$, d and f ($I_2$=7).

*Robustness.* The workflow pool and agent are specific to workflow execution ($R_1$=2). Interactions b, $c_2$, d and f also are ($R_2$=4).

*Extensibility.* Adding a new engine type requires to wrap the engine in the agent and to update interactions b and $c_2$ ($E_1$=3). Updating the version of an engine is transparent ($E_2$=0) *[explain why, also for service]* , and integrating a new workflow is done through interactions $a_1$ and $a_2$ ($E_3$=2). Integrating a new infrastructure requires updates in interactions b, $c_1$ and $c_2$ ($E_4$=3).

*Scalability.* New engine instances only require new agents, which is easily automated ($S_1$=0) and by design very suitable for elastic computing. For instance, auto-scaling rules can be implemented to start new agents when the workload in the science gateway exceeds a certain threshold *[ref needed]* . Distributed engines are not available by default ($S_2$=1) and task scheduling is as complex as in any other architecture ($S_3$=0).

*Other features.* Meta-workflows are not available by default ($O_1$=1). Accessing debugging information is not likely to be an issue since the workflow pool could implement specific functions for that ($O_2$=0).

### *4.6. Nested workflows with service invocation*

*Integration.* Setting up a nested workflow architecture with service invocation requires a science gateway, 2 workflow services and 2 infrastructures ($I_1$=5). The architecture involves 11 interactions ($I_2$=11): $a_1$ (twice: once for each infrastructure), $a_2$, b (twice: once for each infrastructure), $c_1$ (twice: once for each infrastructure), $c_2$ (twice: once for each workflow engine and infrastructure) and d (twice: once for each workflow engine).

*Robustness.* The two workflow services are specific to workflow execution ($R_1$=2). Interactions b (counted twice), $c_2$ (twice) and d (twice) also are ($R_2$=6).

*Extensibility.* Adding a new type of *parent* engine requires to implement the corresponding service, to implement interactions b and $c_2$ in the parent engine, and to implement interaction d in the science gateway and in the parent service ($E_1$=5). Adding a new type of *child* engine only requires to implement the corresponding service, to develop interactions b and $c_2$ in the child engine, and to implement interaction d in the parent service ($E_1$=4). We use $E_1$=4.5 in Table 1 to reflect both conditions. Adding a new version in the parent or child engine only requires modifying this engine ($E_2$=0). Adding a new workflow is done through interaction $a_1$ and $a_2$ ($E_3$=2). Adding a new infrastructure requires to re-implement interactions b and $c_2$ twice, and interaction $c_1$ once ($E_4$=5).

*Scalability.* As in the service architecture, adding a new workflow instance requires manual configuration in the science gateway (instance of a parent engine), or in the parent engine (instance of a child engine) ($S_1$=1). Similarly, elastic engines are difficult to achieve. Distributed engines are possible, through meta-workflows ($S_2$=0). Task scheduling is more complex than in other architectures though, due to the fact that workflow execution is split in different engines ($S_3$=1).

*Other features.* Meta-workflows are possible, which is one of the main interest of this architecture ($O_1$=0). Debugging is difficult because the science gateway cannot directly access fine-grained information in the sub-workflow ($O_2$=1).

### 4.7. Workflow import with service invocation

*Integration.* Workflow import with service invocation requires the same components as for service invocation ($I_1$=3). It requires an additional g interaction for workflow import ($I_1$=7)

*Robustness.* Since workflow conversion is not involved in the execution (it is an offline process), metrics are as in the service architecture ($R_1$=1, $R_2$=3).

*Extensibility.* Since adding a new type of workflow engine aims at supporting more workflows, we consider that it only requires to re-implement interaction g in this architecture ($E_1$=1). Note, however, that implementing interaction g can require very substantial work depending on the complexity of the language used by the new engine. Based on the same logic, adding a new engine version only requires modifying interaction g ($E_2$=1). Adding a new workflow is done through interactions $a_1$, $a_2$ and g ($E_3$=3). As in the service architecture, interfacing with a new infrastructure requires modifications in the workflow service and in interactions b and $c_2$ ($E_4$=3).

*Scalability.* Since workflow conversion is not involved in the execution (it is an offline process), metrics are as in the service architecture ($S_1$=1, $S_2$=1, $S_3$=0).

*Other features.* Meta-workflows are available after import, by connecting workflows in the language used in the science gateway ($O_1$=0). Debugging information is accessible as in the service invocation architecture ($O_2$=0).

## 5. Discussion

### 5.1. Comparison between architectures

Tight integration and sub-tasking require the least amount of integration effort, closely followed by service integration and workflow import (with service invocation). Pool is a bit behind but still before nested workflows (with service invocation) which requires more integration than the other architectures. Robustness leads to the same ordering of architectures, with tight integration and sub-tasking in the top group, service integration and workflow import (with service invocation) close behind, pool in a third group, and nested workflows (with service invocation) at the end. This ordering is consistent across metrics, and it reflects the global complexity of the architectures.

Regarding extensibility, most architectures are overall comparable, except nested workflows (with service invocation) which is significantly behind. This is explained by the complexity of the nested workflow architecture, with 2 infrastructures and 2 workflow services. Sub-tasking and workflow import (with service invocation) perform a bit better to integrate new engines ($E_1$), which is interesting given the current profusion of engines with different characteristics. However, sub-tasking, workflow import (with service invocation) and tight integration perform worse than the others to add engine versions ($E_2$) due to the need to update infrastructure (sub-tasking), science gateway (tight integration), or workflow converter (workflow import). Workflow import performs worse than the others to integrate new workflows ($E_3$) because of the language conversion step. All architectures except nested workflows perform the same to integrate new infrastructures ($E_4$).

The pool architecture is overall the most scalable, which is not a surprise since it was designed precisely for scalability. Tight integration is the least scalable and all the other architectures perform the same. The overall scalability score, however, should not mask the unique characteristics of architectures regarding this criterion. For instance, nested workflows are the only architecture that can easily accommodate distributed workflow execution, which can be critical in some cases. At the same time, the scheduling constraints created by the sub-tasking and nested workflow architectures could well become showstoppers depending on the type of targeted infrastructure. Non-reliable infrastructures, for example, could hardly cope with workflow engines being wrapped in computing tasks as done in sub-tasking. The availability of multiple engine instances could also become a critical feature for science gateways with important workloads, which would favor sub-tasking and pool.

Differences in other features should not be neglected. Nested workflows (with service invocation) and workflow import (with service invocation) are the only architectures that support meta-workflows, which may be interesting in some cases. The availability of fine-grained debugging information may also be critical to efficient user support.

Table 2 provides an overall comparison between architectures, based on the metrics in Table 1.

| | Tight | Service | Sub-task | Pool | Nested | Import |
|---|---|---|---|---|---|---|
| **Integration effort** | | | | | | |
| Total components – $\mathtt{I}_1$ | 2 | 3 | 2 | 4 | 5 | 3 |
| Total interactions – $\mathtt{I}_2$ | 5 | 6 | 5 | 7 | 11 | 7 |
| **Total** (total software pieces) | **7** | **9** | **7** | **11** | **16** | **10** |
| **Robustness of workflow execution** | | | | | | |
| Specific components – $\mathtt{R}_1$ | 0 | 1 | 0 | 2 | 2 | 1 |
| Specific interactions – $\mathtt{R}_2$ | 2 | 3 | 1 | 4 | 6 | 3 |
| **Total** (specific software pieces) | **2** | **4** | **1** | **6** | **8** | **4** |
| **Extensibility** | | | | | | |
| New engine type – $\mathtt{E}_1$ | 3 | 4 | 2 | 3 | 4.5 | 1 |
| New engine version – $\mathtt{E}_2$ | 1 | 0 | 1 | 0 | 0 | 1 |
| New workflow – $\mathtt{E}_3$ | 2 | 2 | 2 | 2 | 2 | 3 |
| New infrastructure – $\mathtt{E}_4$ | 3 | 3 | 2 | 3 | 5 | 3 |
| **Total** (difficulty to extend) | **9** | **9** | **7** | **8** | **11.5** | **8** |
| **Scalability** | | | | | | |
| Multiple engine instances – $\mathtt{S}_1$ | 2 | 1 | 0 | 0 | 1 | 1 |
| Distributed engines – $\mathtt{S}_2$ | 1 | 1 | 1 | 1 | 0 | 1 |
| Task scheduling – $\mathtt{S}_3$ | 0 | 0 | 1 | 0 | 1 | 0 |
| **Total** (scalability issues) | **3** | **2** | **2** | **1** | **2** | **2** |
| **Other features** | | | | | | |
| Meta-workflow – $\mathtt{O}_1$ | 1 | 1 | 1 | 1 | 0 | 0 |
| Fine-grained debugging – $\mathtt{O}_2$ | 0 | 0 | 1 | 0 | 1 | 0 |
| **Total** (missing other features) | **1** | **1** | **2** | **1** | **1** | **0** |

Table 1: Architecture evaluation. Lower values (brighter colors) indicate better performance. On each row, metric values are normalized between 0 (best value) and 1 (worst value) to determine the color of the corresponding table cell. More precisely, the normalized metric value $m'$ is defined as $\frac{m-m_{\min}}{m_{\min}-m_{\max}}$ where $m$ is the initial metric value, $m_{\min}/m_{\max}$ are the minimal/maximal values among all architectures. The RGB hexadecimal color code of the cell is `#99XX99`, where X=F-4m'.

| | Tight | Service | Sub-task | Pool | Nested | Import |
|---|---|---|---|---|---|---|
| Integration effort (normalized total) | 0.00 | 0.22 | 0.00 | 0.44 | 1.00 | 0.33 |
| Robustness (normalized total) | 0.14 | 0.43 | 0.00 | 0.71 | 1.00 | 0.43 |
| Extensibility (normalized total) | 0.44 | 0.44 | 0.00 | 0.22 | 1.00 | 0.22 |
| Scalability (normalized total) | 1.00 | 0.50 | 0.50 | 0.00 | 0.50 | 0.50 |
| Other features (normalized total) | 0.50 | 0.50 | 1.00 | 0.50 | 0.50 | 0.00 |
| | **2.1** | **2.1** | **1.5** | **1.9** | **4.0** | **1.5** |

Table 2: Overall evaluation. Brighter colors and lower scores indicate better performance. Scores are obtained by summing the normalized total scores ($m'$ values) of each criterion in Table 1 and the colors are obtained by normalizing these scores as done in Table 1.

While Table 2 provides a global comparison between architectures, it should not be used without the detailed metrics reported on Table 1 and discussed above. Overall, sub-tasking, and workflow import (with service invocation) perform a bit better than tight integration, service invocation and pool. Nested workflows stands a bit behind for the reasons mentioned previously.

### *5.2. Limitations*

A few limitations remain with our evaluation. First of all, the evaluation is done at a quite high-level of abstraction that may be a bit artificial when applied to real systems. Abstracting architectures from their implementation in specific systems is fundamental to the progress of software engineering of science gateways, but we also realize that this comes at the cost of realism.

In particular, the presented architectures are pure patterns that may be blended in actual systems. The distinction between tight integration and service invocation, for instance, may not be that clear in practice. Service invocation may also be combined with sub-tasking in some cases.

Moreover, all the interactions involved in the architectures were treated equally, as if they all required similar amounts of development effort, which is realistically not the case. For example, interaction `g` (workflow language conversion) clearly requires more effort than interaction `d` (service invocation), and may create much more reliability issues as well. However, quantifying the robustness and amount of development associated with each interaction and software component cannot realistically be done without entering into the details of a particular system.

The particular case of interaction `g` (workflow language conversion) should be further discussed since it is very likely that this interaction could not be easily generalized to any workflow language. For instance, converting FSL, PSOM or Nipype pipelines to any other workflow language is very tricky because these engines rely on general-purpose programming languages (bash, Octave, Python, etc) which are much richer than scientific workflow languages.

The abstract nested workflows and workflow import patterns were instantiated with service invocation so that they can be analyzed in the same framework as the other architectures. Other types of instantiation, for instance nested workflows with sub-tasking, could also be envisaged. We chose to limit ourselves to instantiations with service invocation because the resulting architectures are implemented in real systems, and because service invocation is a largely used architecture. Nested workflows with sub-tasking, in particular, could be an interesting architecture to explore.

Finally, there is no need to say that the particular technical or historical context of a science gateway project may influence the choice of an architecture to integrate workflow engines. For instance, some workflow engines may be already available as web services, which would tend to favor service invocation, and other science gateways may have strongly tested task and data control features (interactions `b` and `c`), which would favor sub-tasking. The migration cost between architectures is totally ignored as well.

## 6. Conclusion

We abstracted architectures so that we can compare them using objective metrics. We illustrated these architectural patterns on examples of real systems.

What this work highlights and recommendations for science gateway developers.

## 7. Acknowledgments

*[FLI-IAM, Labex PRIMES, Ludmer Centre, whatever grant is funding POQ for integrating CBRAIN with PSOM.]*

We warmly thank Rafael Ferreira da Silva for implementing the VIP platform and creating Figure 3.

## References

[1] T. Fahringer, R. Prodan, R. Duan, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, M. Wieczorek, Askalon: A grid application development and computing environment, in: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing, IEEE Computer Society, 2005, pp. 122–131.

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

[3] T. Glatard, J. Montagnat, D. Lingrand, X. Pennec, Flexible and efficient workflow deployment of data-intensive applications on grids with moteur, Journal of High Performance Computing and Applications 22 (3) (2008) 347–360.

[4] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, et al., Pegasus: A framework for mapping complex scientific workflows onto distributed systems, Scientific Programming 13 (3) (2005) 219–237.

[5] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. F. da Silva, M. Livny, K. Wenger, Pegasus, a workflow management system for science automation, Future Generation Computer Systems 46 (2015) 17 – 35. doi:http://dx.doi.org/10.1016/j.future.2014.10.008.
URL http://www.sciencedirect.com/science/article/pii/S0167739X14002015

[6] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. Von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, M. Wilde, Swift: Fast, reliable, loosely coupled parallel computation, in: Services, 2007 IEEE Congress on, IEEE, 2007, pp. 199–206.

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

[8] I. Taylor, M. Shields, I. Wang, A. Harrison, The triana workflow environment: Architecture and applications, in: Workflows for e-Science, Springer, 2007, pp. 320–339.

[9] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, H. T. Vo, Managing the evolution of dataflows with vistrails, in: Data Engineering Workshops, 2006. Proceedings. 22nd International Conference on, IEEE, 2006, pp. 71–71.

[10] P. Kacsuk, Z. Farkas, M. Kozlovszky, G. Hermann, A. Balasko, K. Karoczkai, I. Marton, WS-PGRADE/gUSE Generic DCI Gateway Framework for a Large Variety of User Communities, Journal of Grid Computing 10 (4) (2012) 601–630.

[11] K. Gorgolewski, C. D. Burns, C. Madison, D. Clark, Y. O. Halchenko, M. L. Waskom, S. S. Ghosh, Nipype: a flexible, lightweight and extensible neuroimaging data processing framework in python, Front Neuroinform 5 (3).

[12] P. Bellec, S. Lavoie-Courchesne, P. Dickinson, J. Lerch, A. Zijdenbos, A. C. Evans, The pipeline system for Octave and Matlab (PSOM): a lightweight scripting framework and execution engine for scientific workflows, Frontiers in neuroinformatics 6 (2012) 7.

[13] J. Ashburner, SPM: a history, NeuroImage.

[14] M. Jenkinson, C. F. Beckmann, T. E. Behrens, M. W. Woolrich, S. M. Smith, {FSL}, NeuroImage 62 (2) (2012) 782 – 790, 20 {YEARS} {OF} fMRI20 {YEARS} {OF} fMRI. doi:http://dx.doi.org/10.1016/j.neuroimage.2011.09.015.
URL http://www.sciencedirect.com/science/article/pii/S1053811911010603

[15] S. Reynaud, Uniform access to heterogeneous grid infrastructures with JSAGA, in: Production grids in Asia, Springer, 2010, pp. 185–196.

[16] Á. Hajnal, Z. Farkas, P. Kacsuk, Data avenue: remote storage resource management in WS-PGRADE/gUSE, in: 6th International Workshop on Science Gateways (IWSG), IEEE, 2014, pp. 1–5.

[17] S. Olabarriaga, T. Glatard, A. Hoheisel, A. Nederveen, D. Krefting, Crossing HealthGrid Borders: Early Results in Medical Imaging, in: HealthGrid'09, Berlin, 2009.

[18] K. Plankensteiner, J. Montagnat, R. Prodan, IWIR: A Language Enabling Portability Across Grid Workflow Systems, in: Workshop on Workflows in Support of Large-Scale Science (WORKS'11), Seattle, USA, 2011.

[19] V. Ardizzone, R. Barbera, A. Calanducci, M. Fargetta, E. Ingrà, I. Porro, G. La Rocca, S. Monforte, R. Ricceri, R. Rotondo, D. Scardaci, A. Schenone, The DECIDE Science Gateway, Journal of Grid Computing 10 (4) (2012) 689–707.

[20] M. A. Miller, W. Pfeiffer, T. Schwartz, Creating the cipres science gateway for inference of large phylogenetic trees, in: Gateway Computing Environments Workshop (GCE), 2010, IEEE, 2010, pp. 1–8.

[21] I. D. Dinov, J. D. Van Horn, K. M. Lozev, R. Magsipoc, P. Petrosyan, Z. Liu, A. MacKenzie-Graham, P. Eggert, D. S. Parker, A. W. Toga, Efficient, distributed and interactive neuroimaging data analysis using the LONI pipeline, Frontiers in Neuroinformatics 3.

[22] S. Marru, L. Gunathilake, C. Herath, P. Tangchaisin, M. Pierce, C. Mattmann, R. Singh, T. Gunarathne, E. Chinthaka, R. Gardler, et al., Apache airavata: a framework for distributed applications and computational workflows, in: Proceedings of the 2011 ACM workshop on Gateway computing environments, ACM, 2011, pp. 21–28.

[23] D. Szejnfeld, P. Dziubecki, P. Kopta, M. Krysinski, T. Kuczynski, K. Kurowski, B. Ludwiczak, T. Piontek, D. Tarnawczyk, M. Wolniewicz, P. Domagalski, J. Nabrzyski, K. Witkowski, Vine Toolkit - Towards portal based production solutions for scientific and engineering communities with grid-enabled resources support, Scalable Computing: Practice and Experience 11 (2).

[24] T. Glatard, C. Lartizien, B. Gibaud, R. Ferreira da Silva, G. Forestier, F. Cervenansky, M. Alessandrini, H. Benoit-Cattin, O. Bernard, S. Camarasu-Pop, N. Cerezo, P. Clarysse, A. Gaignard, P. Hugonnard, H. Liebgott, S. Marache, A. Marion, J. Montagnat, J. Tabary, D. Friboulet, A virtual imaging platform for multi-modality medical image simulation, IEEE Transactions on Medical Imaging 32 (1) (2013) 110–118.

[25] P. Kacsuk, Science Gateways for Distributed Computing Infrastructures, Springer, 2014.

[26] T. Sherif, P. Rioux, M.-E. Rousseau, N. Kassis, N. Beck, R. Adalat, S. Das, T. Glatard, A. Evans, CBRAIN: A web-based, distributed computing platform for collaborative neuroimaging research, Frontiers in Neuroinformatics 8 (54).

[27] T. Glatard, P. Quirion, R. Adalat, N. Beck, R. Bernard, B. Caron, Q. Nguyen, P. Rioux, M.-E. Rousseau, A. Evans, P. Bellec, Integration between PSOM and CBRAIN for distributed execution of neuroimaging pipelines (2016).

[28] M. Turilli, M. Santcroos, S. Jha, A comprehensive perspective on the pilot-job abstraction, arXiv preprint arXiv:1508.04180.

[29] D. Rogers, I. Harvey, T. Huu, K. Evans, T. Glatard, I. Kallel, I. Taylor, J. Montagnat, A. Jones, A. Harrison, Bundle and pool architecture for multi-language, robust, scalable workflow executions, Journal of Grid Computing 11 (3) (2013) 457–480.

[30] G. Terstyanszky, T. Kukla, T. Kiss, P. Kacsuk, Á. Balaskó, Z. Farkas, Enabling scientific workflow sharing through coarse-grained interoperability, Future Gener-

ation Computer Systems 37 (2014) 46–59.
[31] K. Plankensteiner, R. Prodan, M. Janetschek, T. Fahringer, J. Montagnat, D. Rogers, I. Harvey, I. Taylor, Á. Balaskó, P. Kacsuk, Fine-Grain Interoperability of Scientific Workflows in Distributed Computing Infrastructures, Journal of Grid Computing (JOGC) 11 (3) (2013) 429–456.