D4.2
Execution Framework Prototype

Work package	WP4High-level Middleware	
Author(s)	Francois Tessier Jayesh Badwaik Salem El Sayed	ETHz JUELICH JUELICH
Reviewer #1	Simon Smart	ECMWF
Reviewer #2	Dirk Pleiter	JUELICH
Dissemination Level	public	
Nature	other	

Date	Author	Comments	Version	Status
27.07.2020	Salem El Sayed		V0.1	Draft
29.07.2020	Francois Tessier		v0.2	Draft
03.08.2020	Jayesh Badwaik		v0.3	Draft
04.08.2020	Salem El Sayed	New template	v0.4	Draft
12.08.2020	Salem El Sayed	Internal review	v0.5	Draft
20.08.2020	Francois Tessier		v0.6	Draft
24.08.2020	Jayesh Badwaik		v0.7	Draft
29.08.2020	Salem El Sayed	Final Checks	v1	Final

DATA ORCHESTRATION IN HIGH PERFORMANCE COMPUTING
This project has received funding from the European Union's Horizon 2020 research and innovation program through grant agreement 801101.

Executive Summary

The following report is part of D4.2 and details the software release of the execution framework prototype developed as part of the Maestro project. The work included is part of WP4 which focuses on high-level middleware.

Contents

Glossary

CDO	...	Core Data Object
MIO	...	Maestro I/O interface
PM	...	Pool Manager
CWL	...	Common Workflow Language
DAG	...	Directed Acyclic Graph
DynPro	...	Dynamic Provisioning

1 Introduction

A workflow consists of set of applications which are dependent on each other by the way of data produced and consumed by those applications. In any kind of workflow, there is a control flow which indicates when an application can start and triggers events when it has ended. This is what we denote by **execution flow**. On the other hand, there is the flow of data between applications. This indicates how data flows between producer and consumer applications in the workflow and includes the data transfer system. We denote this flow of data and its dependencies in a workflow as **data flow**. Depending on the situation, the workflow manager and the different applications involved may or may not be aware of these data transfers and their dependencies. Even in cases where the components of the workflow are aware about the data flow, the information might only be very partial or local.

An example of partial information is where the dependency of one data on another is specified in terms of execution flow. For example an application `B` might only start after application `A` has finished executing successfully and has produced some file system artifact. Such scenarios are quite common in workflow managers like Pegasus [1] where end of processes are the edge triggers. It might completely be possible `A` actually does not produce any data and there is no actual data dependency between `B` and `A`. A second possible scenario here is that `A` writes to some global store and `B` reads from such a global store with the workflow manager having no information about the state of global store. The information is considered partial because while the dependence of `B` on `A` is globally known, the precise information about dependence of data is not known.

An example of local information is where applications transfer data between each other without the workflow manager or any other monitor application being aware of the data transfer. One can see this, for example, in programs like TerrSysMP' [2] which uses a coupler to transfer data between different applications. Here the complete dependency of the data between applications is known only by the applications themselves. Even then, the applications might only know their own part in the data dependency without knowing the complete context. Furthermore, the workflow manager has no information about the data transfer, and therefore cannot make any decisions to the workflow execution based on the state of data transfer. This is why the information is called local, since the complete information about the data transfer is available in the system, but it is only available locally to each application.

The lack of complete knowledge of data flow can introduce inefficiencies in the workflow execution. These inefficiencies generally take two forms, a scheduling inefficiency and a resource inefficiency. A scheduling inefficiency happens when an action is not scheduled to start even after all the data dependencies of the action have been satisfied. One can see this inefficiency in the partial information example mentioned previously. The application `B` can start as soon as the application `A` produces the data required. However, due

to the partial information available, it will wait till A has itself finished which might be much after the availability of data.

A resource inefficiency happens when there is a mismatch between the format and/or timing of output of one application and the format of input for another application leading to extraneous copies. There are two scenarios which illustrate this kind of inefficiencies in a workflow. The first scenario is when the output format of the produced data is different from the input data format required by the consumer. In Maestro framework, the Maestro core takes care of this problem, and provides the data in a uniform format for all the applications.

The second scenario is when an application, having produced and offered the data for consumption, now wants to withdraw the data offering and the consumer applications is not yet ready to receive it. The situation can generally be dealt with in two ways. One method is to create a copy of the data in a temporary store which can then be offered to the consumer later. The second method is to block the producer application till the consumer has consumed the data. There is no single correct answer for this problem, and the actual answer might depend on a large number of factors. In this document, we only provide a rudimentary solution and defer the exploration of complete solution to D4.4. Even though the solution is rudimentary, it should suffice for most of the cases.

Maestro aims to solve both these kinds of efficiencies by introducing data awareness into the execution flow and feeding it into workflow execution. However, most workflow managers are not designed for exchanging their underlying data flow systems. In order to plug this deficiency, we have introduced an execution framework which can complement the Maestro core middleware in providing efficient execution flow and data flow to the applications.

This document is a report highlighting some of the design details of the execution framework. It's target is to document the released prototype software. The remainder of this document details the execution framework prototype released as part of D4.2. Section 2 provides a data-aware view of workflows and lays out the context for the discussion of further topics. Section 3 contains a description of the overall execution framework design. Meanwhile, section 4 includes information on the software released.

1.1 Relation to other parts of the project

In general WP4 is tasked with developing high-level middleware to facilitate the use of Maestro core. Part of this effort is supporting workflow execution by developing a framework that augments the execution with Maestro required components. The work involved requires bridging the gap between workflow requirements as investigated in WP2 with the Maestro middleware developed in WP3 [3]. The handling of these requirements by Maestro core as part of WP3 versus handling them by the workflow execution as part of WP4, was discussed in D2.3 [4]. Additionally, our understanding of workflow

semantics has been furthered by the use-cases studied in WP2 and on which some of our workflow categorisation discussed in this deliverable has been based.

In later work as part of WP4, the here detailed execution framework prototype is developed further. The final version is then included as a demonstrator delivered as an outcome of WP6.

2 Data Flows: Analysis and Classification

In this section, we describe the different kind of data flows that might be possible in a workflow, and set the context for the various different components of execution framework that will be detailed in the following sections. Generally, when talking about workflows and workflow managers, the focus is more on execution flow and resource allocation. Data flow is generally ignored, and if treated, is only treated as a part of execution flow. In order to properly track the dependencies of data, it is important to concentrate on the data flow instead of the execution flow. Situations where the execution depends on the data flow can then be handled as extra effects of the data flow analysis.

One can think of data flow as a directed graph, with each instance of data being the nodes in those graph. The data flow graph obtained is a directed acyclic graph. In Maestro, each data object is represented by a CDO which are then identified uniquely by a name. For the rest of the discussion, we will use a CDO to denote data produced in an application for better clarity of discussions and to avoid overusing the word data in multiple contexts. An example of a graph of data flow can be seen in Figure 1. There are three applications in the workflow. The CDOs produced by the applications are color coded. So, application `A` produces CDOs 1 and 3, while the application `B` produces CDOs 2, 4 and 5 and finally application `C` produces CDO 6.

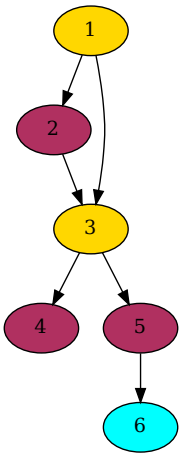

Figure 1: An example workflow's data flow, where application `A` produces CDOs 1 and 3 application `B` produces CDOs 2, 4 and 5 and application `C` produces CDO 6

There are a few aspects and behavior of this data flow which we wish to point out. One can see that CDO 2 in application `B` depends on CDO 1. There are two choices that the application developer for `B` or the workflow developer can make at this point. The first choice is to start the application `B` only when the CDO 1 is available and the second choice is to start the application

and then wait for the CDO. Both are acceptable behaviors and the decision of which one of the two is an appropriate behavior will be determined by either the workflow developer or the application developer. Maestro workflows have to support both the behaviors. More specifically, Maestro has to be able to support modifications to the execution flow that will be required when the first choice is being exercised.

The second aspect we would like to point out is in the situation where CDO 4 depends on CDO 3. Consider the case where we a-priori know that CDO 3 and CDO 4 are going to be produced in the workflow and that CDO 4 depends on CDO 3. In such cases, one can have a workflow manager process monitor the production of CDO 3 and inform application `B` about it's availability to produce CDO 4. However, there are other cases where, such an information might not always be available. Consider a physics simulation which has dynamic time-stepping. At each time step, the simulator `A` outputs a CDO which is then consumed by the post processor. The number of such time steps are dynamic and hence are the number of data produced. So, while there is a semantic relation which shows dependence of CDO 4 and CDO 3, the existence of the data and the relation itself is not a priori established.

Finally, consider the case of resource inefficiency introduced due to timing mismatch where application `A` produces 5 and wants to exit. In such cases, it can often happen that `C` is not yet ready to receive 5. This can happen, for example, when its resources are not completely initialized or are busy in some other process. One of the goals of Maestro project is to not make the workflow intrusive to the application. With that in mind, there are two choices available to the workflow developer. The first choice is to block `A` temporarily in order to allow `C` some breathing space to get ready to receive the resource. The second option is to create a copy of the data and then offer `C` the copy. Due to extra factors in the equation like the availability of the resources on an oft-shared compute node, the working of the workflow managers and the job scheduler itself, there is no unconditionally correct answer.

All the different aspects pointed to above show a lack of information in the graph shown in Figure 4. To address the first two aspects, we add two more annotations to the CDOs, namely a Static/Dynamic annotation and a Execution/Normal annotation. A CDO is annotated with a Static (S) tag if the existence and identity of the CDO is known to the workflow. Otherwise, the CDO is annotated with a Dynamic (D) tag. In the Maestro world, a CDO is Static if both its existence and its name is known to the workflow manager before the start of simulation. If a particular CDO dependency affects the execution flow of the workflow, we annotate the dependency edge with the "Execution" tag, otherwise we use the "Normal" tag.

We would like to point out that two CDOs can both be Dynamic and still be connected with a Execution dependency. However, this can never happen in a static workflow (where the workflow is fully determined before the workflow even starts execution). Consider an extreme case in a static workflow, where you have a simulator producing a CDO node whose name is not yet known since the CDO nodes are named by the index of the time step and

the simulator uses dynamic time stepping. The final time step produces a CDO node which is now to be consumed by the post processing step. In this scenario, one might think that the CDO nodes are dynamic with a Execution dependency even in a static workflow. However, this is incorrect. For the postprocessing application to know which CDO node to request for postprocessing, there needs to be a pre-defined method of transmission of the name of that postprocessing node. We claim that this message is the actual CDO dependency between the two applications, and hence is static in nature.

Handling the third aspect, namely the resource inefficiency due to timing mismatch, is a more tricky problem. This case is especially troublesome, because often, a workflow may well land into this situations due to factors outside of its control. Therefore, one might not know about such a situation till it actually occurs in a workflow. As already noted, there is no unconditionally correct answer for the question. We observe that, for now, one can achieve a correct behavior of the workflow by mandating either an always copy or an always blocking policy, while still not hindering further developments in optimizing this workflow in any way. Hence, for now, we defer the detailed study of these particulars to the future optimization efforts of the execution framework [1]. The solution there will have to be based on using a monitor process which actively tracks the state of the workflow manager as well as the data flow using the subscriber CDO feature of the Maestro core, and makes decisions accordingly. For now, we annotate the edges involved in such a flow with A (Awaiting) and edges which do not suffer from such an issue as F (Fulfilled).

With the above information, a modification of Figure 1 will look as shown in Figure 2 and we are finally in a position to formally define a data flow for our purpose:

Definition 1 (Data Flow)**.** *A data flow is a direct acyclic graph of nodes, where each node can be identified by a label $a(C)$ where a is a unique ID and C can either be S for Static or D for Dynamic. And each edge represents a dependency and is indicated with a label (W, T) where W can either be E for Execution or N for Normal, and T can either be A for Awaiting or F for Fulfilled.*

In the context of Maestro, each node is a CDO. Writing the data flow as a graph allows us to convert the problem of managing workflows into a more tractable problem about graph analysis. We note that this graph is an *a posteriori* view of the workflow. The objective therefore is to devise a strategy which will work for all such graphs. One of the first things to notice is that we can reduce all complicated data flows into a producer consumer model, which is shown in Figure 3. As long as every single case of producer consumer here is handled correctly, the complete graph can be handled correctly.

As seen in Figure 3, there is a binary parameter associated with each of the two nodes and two binary parameters associated with the edge. In total, this gives us 16 different data flow cases. We consider all the cases one by one and prescribe the action to take for each test case. For cases where a more

[1] The documentation of optimization efforts will be done in deliverable D4.5 "Workflow Management and Optimisation Final Release"

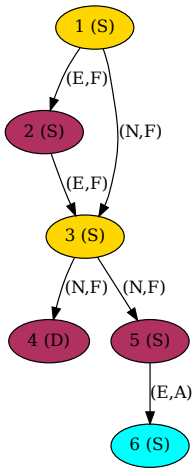

Figure 2: An annotated example workflow's data flow, where application A produces CDOs 1 and 3 application B produces CDOs 2, 4 and 5 and application C produces CDO 6

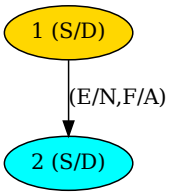

Figure 3: A producer consumer model workflow's data flow

optimal solution might be possible with more information, we indicate those cases as such, and defer them to the future optimisation work.

We denote each case as a 4-tuple, C_p, C_c, W, T, where C_p indicates the value of the C parameter of the producer node, C_c indicates the value of the C parameter of the consumer node, and W, T indicate the (W, T) parameters of the edge. Multiple cases might have the same advice, however, we still indicate them separately for sake of tractability of handling the cases.

Furthermore, the T parameter of the workflow is known only at the point where the situation actually arises. In the following, we will see that only the Awaiting (A) case of the T parameter needs extra attention. Hence, a workflow developer should always prepare for the A case, knowing that the Fulfilled (F) case will be satisfied automatically.

2.1 Analysis of Different Cases

2.1.1 S, S, N, F

In this case, both the CDOs nodes are known to the workflow manager. There is no effect of CDOs on the execution flow. And the state of data is fulfilled. Maestro and the workflow handles this case automatically with no additional action required.

2.1.2 S, S, N, A

In this case, both the CDOs are known to the workflow manager. There is no effect of the CDOs on the execution flow. The state of data is **not fulfilled**. Improving this case is a problem that can be deferred to the future optimisation work. For now, we extend the lifetime of the CDO producer till the consumer has consumed it.

2.1.3 S, S, E, F

In this case, both the CDOs are known to the workflow manager. And the state of data is fulfilled. Maestro handles this case automatically with no additional action required. However, the data availability affects the execution flow. In this case, we use the workflow translator which will be described in Section 3.3. The workflow translation will add extra tasks to the workflow to make sure that the consumer application is started after the data is available.

2.1.4 S, S, E, A

In this case, both the CDOs are known to the workflow manager. However, the data availability affects the execution flow. The state of the data is **not fulfilled**. One part of the problem (the execution part) is solved in a manner similar to S, S, E, F in Section 2.1.3 using a workflow translation. The complete solution to the problem of resource inefficiency is deferred to the future optimisation work. For now, we use a similar method as used for S, S, E, F given in Section 2.1.3 and extend the lifetime of producer data till the consumer has consumed it.

2.1.5 S, D, N, F

In this case, the source CDO is known to the workflow manager. There is no effect of CDOs on the execution flow. And the state of data is fulfilled. Maestro and the workflow handles this case automatically with no additional action required. Note that the consumer CDO information is not known to the workflow manager, but because the data is Fulfilled, the consumer node can be created without any issues.

2.1.6 S, D, N, A

In this case, the source CDO is known to the workflow manager, but the future demand is not known. There is no way for the workflow or the application to know whether the CDO produced will or will not be required in the future in this scenario. Therefore, we do not support this scenario directly in the workflow. If an application developer or the workflow developer suspects that this might be the case, they can mark such a resource as precious or use similar functionality from the Maestro core to make sure that the data is available when needed, reducing the case to S, D, N, F.

2.1.7 S, D, E, F

In this case, the source CDO is known to the workflow manager. The workflow manager already knows that the consumer CDO depends on a source CDO, but it does not know the name of the consumer CDO. Here, the workflow manager knows that there will be a source CDO which will be required to fulfill the consumer CDO. In this case, the workflow manager can create a proxy CDO as soon as the consumer CDO is declared. The proxy CDO has a static name, and is designated to be consumed only when the source CDO itself is consumed by the consumer CDO. In this way, we have broken down the problem into two problems. The dependency relation between source and proxy CDO is a S, S, E, F relation, while the relation between proxy CDO and consumer CDO is a S, D, N, F relation.

2.1.8 S, D, E, A

In this case, the source CDO is known to the workflow manager, but the future demand is not known. There is no way for the workflow or the application to know whether the data node produced will or will not be required in the future. Therefore, we do not support this scenario directly in the workflow. If an application developer or the workflow developer suspects that this might be the case, they can mark such a resource as precious or use similar functionality from the Maestro core to make sure that the data is available when needed, which will reduce this case to S, D, E, F case.

2.1.9 D, S, N, F

In this case, the consumer CDO is known to the workflow manager. The workflow manager already knows that the consumer CDO depends on a source CDO, but it does not know the name of the source CDO. But since the state of the data is fulfilled, Maestro and the workflow handles this case automatically with no additional action required.

2.1.10 D, S, N, A

In this case, the consumer CDO is known to the workflow manager. The workflow manager already knows that the consumer CDO depends on a source CDO, but it does not know the name of the source CDO. Now here, the workflow manager knows that there will be a source CDO which will be required to fulfill the consumer CDO. In this case, the workflow manager can create a proxy CDO as soon as the source CDO is available. The proxy CDO has a static name, and is designated to be consumed only when the source CDO itself is consumed by the consumer CDO. In this way, we have broken down the problem into two problems. The dependency relation between source and proxy CDO is a D, S, N, F relation, while the relation between proxy CDO and consumer CDO is a S, S, N, A relation.

2.1.11 D, S, E, F

In this case, the consumer CDO is known to the workflow manager. The workflow manager already knows that the consumer CDO depends on a source CDO, but it does not know the name of the source CDO. But since the state of the data is fulfilled, Maestro handles this case automatically with no additional action required. However, the data availability affects the execution flow. In this case, we use the workflow translator, which will be described in Section 3.3, to add extra tasks to the workflow to make sure that the consumer application is started only after the data is available. But due to the dynamic nature of the source CDO, the translation is not a straight forward process, and there is some preprocessing required. Specifically, one needs to create a proxy CDO of a static name as soon as the source CDO is created. The relation between the source CDO and the proxy CDO is of the type D, S, N, F. Now, the proxy CDO and the consumer CDO are connected by a S, S, E, F relation, which is previously handled in Section 2.1.3.

2.1.12 D, S, E, A

In this case, the consumer CDO is known to the workflow manager. The workflow manager already knows that the consumer CDO depends on a source CDO, but it does not know the name of the source CDO. Here, the state of the data is not fulfilled and the data availability affects the execution flow. In this case, we use the workflow translator, which will be described in Section 3.3, to add extra tasks to the workflow to make sure that the consumer application is started only after the data is available. But due to the dynamic nature of the source CDO, the translation is not a straight forward process, and there is some preprocessing required. Specifically, one needs to create a proxy CDO of a static name as soon as the source CDO is created. Furthermore, we designate the proxy CDO as consumed only when the source CDO is consumed. The relation between the source CDO and the proxy CDO is of the type D, S, N, F. Now, the proxy CDO and the consumer CDO are connected

by a S, S, E, A relation which is previously handled in Section 2.1.4.

2.1.13 D, D, N, F

In this case, none of the CDOs are known to the workflow manager. There is no effect of CDOs on the execution flow. And the state of data is fulfilled. Maestro and the workflow handle this case automatically with no additional action required. Note that the consumer CDO information is not known to the workflow manager, but because the data is Fulfilled, the consumer can be created without any issues.

2.1.14 D, D, N, A

In this case, the source CDO is known to the workflow manager, but the future demand is not known. There is no way for the workflow or the application to know whether the CDO produced will or will not be required in the future. Therefore, we do not support this scenario directly in the workflow. If an application developer or the workflow developer suspects that this might be the case, they can mark such a resource as precious or use similar functionality from the Maestro core to make sure that the data is available when needed, which will convert this case to a D, D, N, F case.

2.1.15 D, D, E, F

In this case, both CDOs are not known to the workflow manager. The workflow manager already knows that the consumer CDO depends on a producer CDO, but it does not know the name of either of the CDOs. This problem can be solved by another level of indirection, where an additional source and consumer CDOs are created and the name of those CDOs is fixed a priori. These additional CDOs contain the name of the original CDOs. The relation between the original source CDO and the additional source CDO is a D, S, N, F relation, while the relation between the additional source CDO and additional consumer CDO is now a S, S, E, F relation. Finally, the relation between the additional consumer CDO and the consumer CDO is a S, D, N, F relation.

2.1.16 D, D, E, A

In this case, the source node is known to the workflow manager, but the future demand is not known. There is no way for the workflow or the application to know whether the data node produced will or will not be required in the future. Therefore, we do not support this scenario directly in the workflow. If an application developer or the workflow developer suspects that this might be the case, they can mark such a resource as precious or use similar functionality from the Maestro core to make sure that the data is available when needed, which will convert this case to a D, D, E, F case.

2.2 Example Workflow Prototype

Out of the above workflows, the non trivial ones are the ones where additional software is required to make the workflow succeed. The list of such workflows is given below:

1. S, S, N, A
2. S, S, E, F
3. S, S, E, A

We study these workflows as examples in the example workflow software code (`maiti`), which we released as part of this deliverable. The details of the software release are given in Section 4.

3 Execution Framework Components

The execution of a Maestro-enabled workflow will require a combination of newly developed components, enhancement of existing components and the use of unmodified common elements of an HPC system. Figure 4, depicts those components and how they are articulated together. The rest of this section gives the role of each item and shortly describes how we propose to enhance them if this is necessary. Later in this document, we will provide more details.

3.1 Workflow Manager: Pegasus

The workflow landscape includes hundreds of workflow managers for a wide range of solutions and target audience. These differ in many aspects including workflow description language, execution order and target systems. Workflows are hardly transferable between the workflow frameworks and managers without in many cases requiring large porting efforts. There have been some attempts in creating a common structure for describing workflows, notably the Common Workflow Language (CWL) [5]. However these have not seen wide scale adoption by the existing workflow managers.

Since the execution framework designed here requires the presence of a target workflow manager, a selection criteria had to be investigated. In addition to running some tests on a selected number of workflow managers, we relied on the available literature detailing the workflow manager landscape and comparing their characteristics [6] [7] [8] [9]. The criteria were reduced to 3 aspects detailed along with the general assessment in Table 1. Using the criteria various available workflow managers were investigated. This included Swift/T, Parsl, Ecflow, UNICORE, Kepler and Pegasus.

Our selection process showed that Pegasus is the best candidate for use as a proof of concept in the execution framework prototype. This is mainly due to the wide usage of Pegasus in various scientific communities, allowing

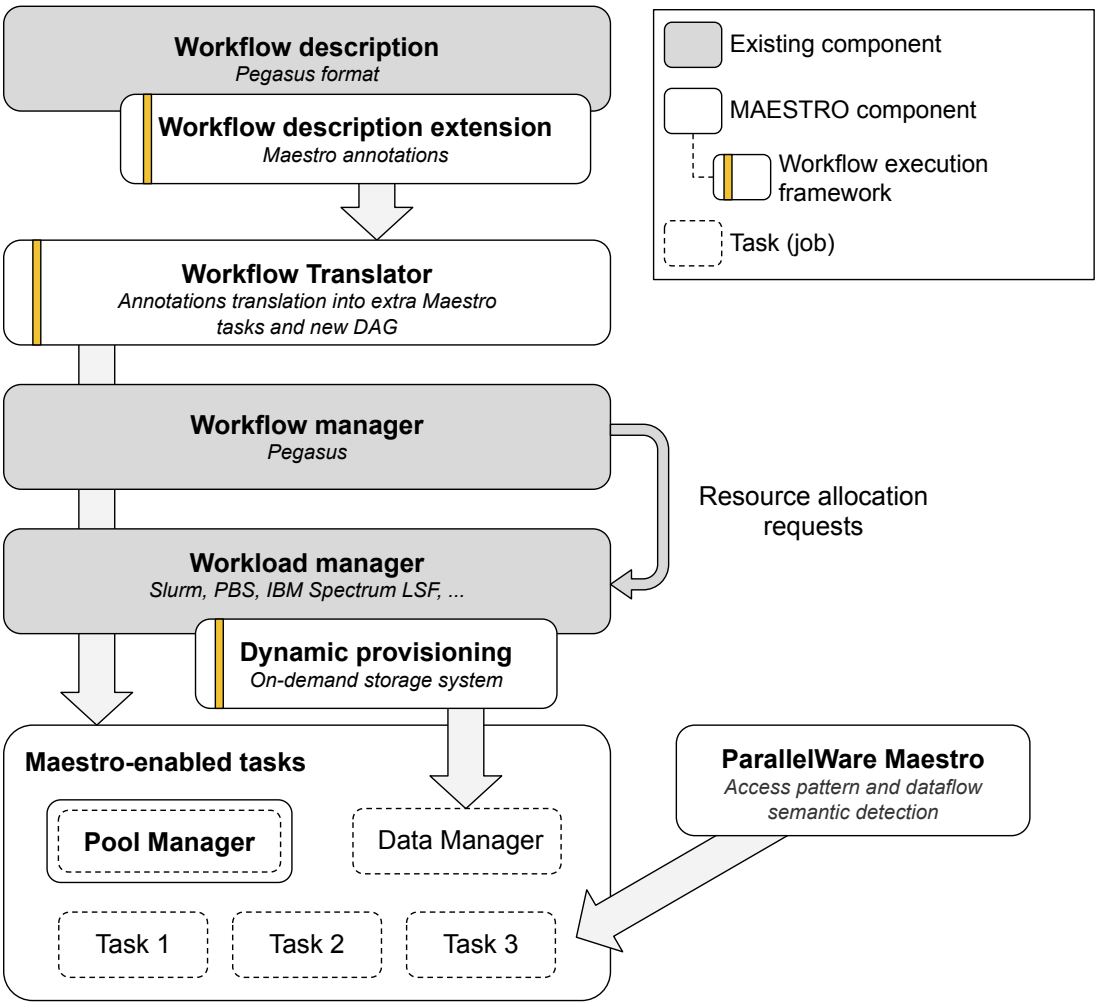

Figure 4: Execution Framework

for testing a wide range of workflow scenarios. Table 2 details the assessment of Pegasus based on the selection criteria.

It is important to note that it is possible to support other workflow managers besides Pegasus by replicating some of the efforts detailed in the execution framework prototype. The required effort highly depends on the target workflow manager.

3.2 Maiti : A Collection of Example Workflows

In Section 2, we talked about different kinds of data flows that were possible, and how they relate to different workflows. There, we noted that analysis of a set of those data flows was enough to prove the correctness of the approach to any generic data flow. `maiti` is a proof of concept of those use cases. For the case of execution flow, `maiti` uses the workflow translator to produce workflow XMLs. For other cases, where the application is supposed to be temporarily blocked, `maiti` uses files to indicate the status of the CDOs. Future

Table 1: Workflow manager criteria and general assessment

Criteria	**General Assessment**
Used in HPC Context	Few workflow managers are widespread in HPC communities and have adequate support for necessary toolchains such as job batch systems (ex. Slurm).
Good uptake	Many workflow managers are designed for and by a specific research community and is hardly used outside that user base.
Flexibility, extendability and usability	Most workflow managers use only file systems for data transfer making it difficult to support different data movement methods. Furthermore, many workflow managers are graphically based which hinders automated workflow extendability.

Table 2: Pegasus assessment according to the selection criteria

Criteria	**Pegasus Assessment**
Used in HPC Context	A large number of execution environments are supported including many HPC systems.
Good uptake	Used by multiple different scientific communities.
Flexibility, extendability and usability	• Uses XML format to describe workflows allowing for parsing and adjusting workflow externally. • As a disadvantage, the sole dependence on file systems for data transport and task controlling has to be managed by other components in the execution framework.

development of the execution framework will have `maiti` use subscription to CDO events, which is supported by Maestro core, to remove dependency on files.

3.3 Workflow Translator

To make an existing workflow ”Maestro-enabled”, we introduced a new component called the “workflow translator”. Based on requirements expressed through annotations, the Workflow Translator takes as input a workflow de-

scription (so far, we support the XML format of Pegasus [1, 10]) with annotations and gives as output a new workflow description augmented with Maestro tasks. Those tasks are inserted in the original graph of tasks to be scheduled. An example illustrating the main concept of the Workflow Translator is given in Figure 5: a workflow description using the Pegasus format and in which we have added Maestro annotations is given as an input parameter to the Workflow Translator. This component abstracts the workflow in the form of a DAG of tasks and parses the annotations. Based on these, it creates new tasks and inserts them in the existing DAG. The new graph is then exported to a new Pegasus-compatible workflow description file.

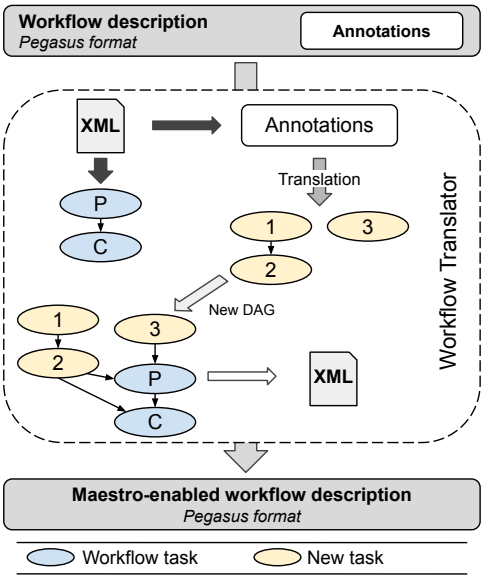

Figure 5: Workflow translator

The annotations supported by the workflow translator are used to express the needs of the workflow, both in terms of Maestro components and resources. Those requirements are qualitative : type of storage backend, persistency, I/O intensive, and so on. The current list of requirements is at an initial stage and will be extended in a near future according to workflow needs. No user-defined characteristics will be allowed in the first version as one or several decision models will rely on those annotations. In the future, we could imagine custom requirements along with the appropriate decision algorithm.

The annotations describing the workflow requirements can take place at two different levels: at a workflow level or at a task level. Typically, we will consider that the main annotation making an existing workflow "Maestro-enabled" will be applied at a workflow level.

Below is an example of a simple Pegasus workflow description for a producer and consumer use-case with two supported annotations (in red). The first annotation (*maestro.workflow.control.enabled="true"*) is the main one in a sense that the tasks required by the Maestro middleware to operate are inserted in the graph of tasks through it. The same-level second annotation (*maestro.workflow.core.backend="minio"*) implies the dynamic deployment of

an object-store (MinIO [11] in this case) as a storage backend of the core of our middleware (See section 2.4 about the dynamic deployement).

```
<?xml version="1.0" encoding="UTF-8"?>
<adag name="mockup" jobCount="2" fileCount="2" childCount="0"
      maestro.workflow.control.enabled="true" maestro.workflow.core.backend="minio">
    <job id="producer" namespace="mockup" name="DataProducer" runtime="35" cores="1">
        <uses file="prod_out.dat" link="output" size="4096"/>
    </job>
    <job id="consumer" namespace="mockup" name="DataConsumer" runtime="35" cores="1">
        <uses file="prod_out.dat" link="input" size="4096"/>
    </job>
</adag>
```

The workflow translator will parse this workflow description and add tasks according to the different annotations. From the given example, all the extra tasks required by Maestro to operate will be added as well as a task for the dynamic provisioning of an object-store. The new workflow description given as output and compliant with Pegasus is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<adag childCount="1" fileCount="2" jobCount="8" name="mockup"
      maestro.workflow.control.enabled="true" maestro.workflow.core.backend="minio">
    <job cores="1" id="producer" name="DataProducer" namespace="mockup" runtime="35">
        <uses file="prod_out.dat" link="output" size="4096" />
    </job>
    <job cores="1" id="consumer" name="DataConsumer" namespace="mockup" runtime="35">
        <uses file="prod_out.dat" link="input" size="4096" />
    </job>
    <job cores="1" id="mstr_pool" name="Maestro Pool Manager" namespace="">
    </job>
    <job cores="1" id="mstr_pmwatcher" name="Pool Manager Watcher" namespace="">
         <uses file="PM.started" link="output" />
    </job>
    <job cores="1" id="mstr_monitor" name="Monitor" namespace="">
         <uses file="PM.started" link="input" />
        <uses file="data.created" link="output" />
        <uses file="monitor.ready" link="output" />
    </job>
    <job cores="1" id="mstr_datawatcher" name="Data Watcher" namespace="">
        <uses file="data.created" link="input" />
    </job>
    <job cores="1" id="mstr_monitorwatcher" name="Monitor Watcher" namespace="">
        <uses file="monitor.ready" link="input" />
    </job>
    <job cores="1" id="mstr_dynpro_minio" name="Dynamically provisioned MinIO" namespace="">
    </job>
    <child ref="mstr_monitor">
        <parent ref="mstr_pmwatcher" />
    </child>
</adag>
```

3.4 Dynamic Provisioning

It is common for HPC systems to provide dynamic access to compute nodes through a batch scheduler. However, little has been done for dynamically provisioned storage resources. Such resources are traditionally shared among all users. A dynamic provisioning mechanism allows to allocate dedicated resources to an application or a workflow and deploy an appropriate data manager on top of it. In the context of Maestro, such a tool can particularly deploy

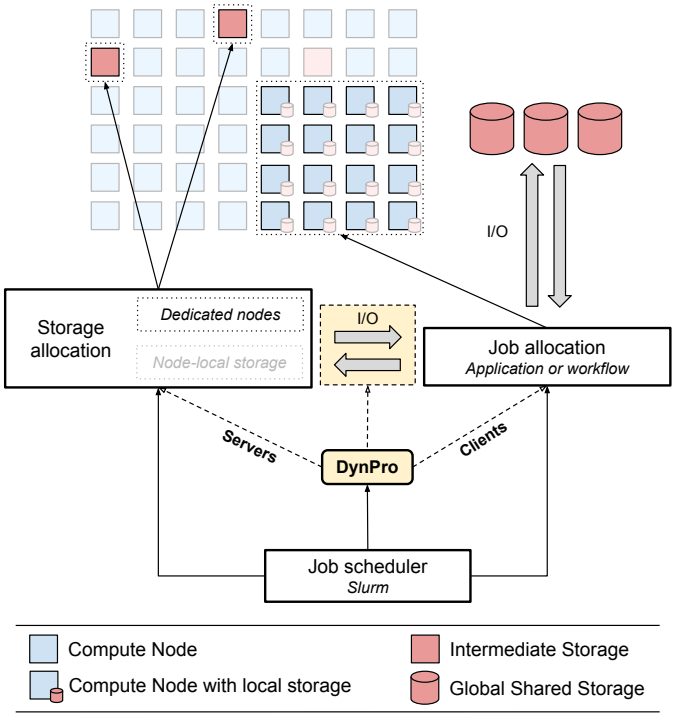

Figure 6: DynPro for dynamically provisioning storage systems on intermediate storage resources

the storage backend required by the core of our middleware or by any other components (telemetry for instance).

Therefore, we introduced an Ansible-powered dynamic storage resource provisioning (DynPro) [12] mechanism that can use intermediate storage to set up on-demand a data manager such as a parallel file system, an object store or a database. The tool takes as input a list of already allocated storage resources and the name of a supported data manger to deploy onto. A task can be created by the workflow translator (see previous section) to set up a data management system using DynPro.

As Ansible [13] has been designed for flexible deployment of services, we use this software to manage clients and servers of the requested data manager. Once a storage allocation has been requested through the batch scheduler, a job-scheduler-agnostic script associated with a configuration file describing the available storage resources of that allocation is in charge of the deployment process. A storage allocation can be the same as the compute allocation (node-local storage) or can be a set of nodes dedicated to storage (burst buffers for instance). To minimize the footprint on the target resources (privileges), all the data managers we support are containerized and launched with a container engine for HPC systems. Figure 6 presents the general functioning of DynPro. For this example, we favor the use of burst buffer nodes (red squares) instead of node-local storage.

Our mechanism, in its version tagged "d4.2", supports multiple data managers such as a parallel file-system (BeeGFS [14]), a database (Cassandra [15])

and an object-based storage system (MinIO [11]), and implements a basic stage-in/stage-out procedure to backup and restore the state of a data manager.

4 Prototype Release

In this section we detail the software released as part of the execution framework prototype. Whenever possible we include some specific commands to compile and/or run the necessary components. More details can be found in the repositories themselves and their included ' `README.md` files.

4.1 Workflow Manager: Pegasus Setup and Configuration

The latest available Pegasus[2] workflow manager can be installed on CentOS 7 using the following set of commands:

```
$ wget -O /etc/yum.repos.d/pegasus.repo \
       http://download.pegasus.isi.edu/wms/download/rhel/7/pegasus.repo
$ yum-config-manager --enable testing
$ yum update
$ yum install pegasus
```

4.2 Maiti: A collection of Workflow Examples

Source code	`https://gitlab.version.fz-juelich.de/maestro/maiti`
Tag	D4.2
Prerequisites	• Maestro core • Boost (1.73)
License	Apache v2

4.3 Workflow Translator

Source code	`https://gitlab.version.fz-juelich.de/maestro/workflow-translator`
Tag	d4.2
Prerequisites	• Python 3
License	BSD 3-clause

The following command is an example of how to use the workflow translator to convert an annotated workflow description to a Maestro-enabled one. The workflow description given as input is provided in the repository:

```
$ ./wf_translator.py -w workflows/LEAD_Data_Mining_Workflow/leaddm.xml -f pegasus -v
```

The full list of options can be extracted using the help argument:

```
$ ./wf_translator.py --help
usage: wf_translator.py [-h] [-w WORKFLOW] [-f FORMAT] [-v]

optional arguments:
  -h, --help            show this help message and exit
```

[2] Currently the latest available Pegasus version which was used for testing is 4.9.3

```
-w WORKFLOW, --workflow WORKFLOW
                      Workflow description file
-f FORMAT, --format FORMAT
                      Workflow manager (Pegasus, ...)
-v, --verbose         Display debug information
```

Running the resulting workflow using Pegasus is done as described in its user guide [10]

4.4 Dynamic Provisioning

Source code	`https://github.com/eth-cscs/dynamic-resource-provisioning`
Tag	d4.2
Prerequisites	Master node: • Python 3 • Virtualenv (ansible, hostlist) Worker nodes: • Sarus, a container engine for HPC • Storage resources (on-node SSD)
License	BSD 3-clause

5 Concluding Remarks

In this document we report on the software released for the execution framework prototype as part of D4.2. The framework allows for executing workflows using the Pegasus workflow manager with support of the Maestro middleware. The optimisation and further development of the prototype will be undertaken as part of future work. The final results will be described in a demonstrator towards the end of the Maestro project.

References

[1] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz, "Pegasus: a framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming Journal*, vol. 13, no. 3, pp. 219–237, 2005.

[2] P. Shrestha, M. Sulis, M. Masbou, S. Kollet, and C. Simmer, "A Scale-Consistent Terrestrial Systems Modeling Platform Based on COSMO, CLM, and ParFlow," *Monthly Weather Review*, vol. 142, pp. 3466–3483, 08 2014.

[3] C. Haine, U. Haus, and A. Tate, "D3.1 Initial Core Middleware Specification, API Document," 2019.

[4] D. Sármány, S. Smart, J. Capul, F. Tessier, and S. El Sayed, "D2.3 Middleware Requirements and API Design," 2020.

[5] "Common workflow language." `https://www.commonwl.org/`. Accessed: 2020-08-11.

[6] M. Atkinson, S. Gesing, J. Montagnat, and I. Taylor, "Scientific workflows: Past, present and future," *Future Generation Computer Systems*, vol. 75, pp. 216 – 227, 2017.

[7] R. Ferreira da Silva, R. Filgueira, I. Pietri, M. Jiang, R. Sakellariou, and E. Deelman, "A characterization of workflow management systems for extreme-scale applications," *Future Generation Computer Systems*, vol. 75, pp. 228 – 238, 2017.

[8] K. Plankensteiner, R. Prodan, M. Janetschek, T. Fahringer, J. Montagnat, D. Rogers, I. Harvey, I. Taylor, Á. Balaskó, and P. Kacsuk, "Fine-grain interoperability of scientific workflows in distributed computing infrastructures," *Journal of Grid Computing*, vol. 11, pp. 429–455, Sep 2013.

[9] J. Arshad, G. Terstyanszky, T. Kiss, N. Weingarten, and G. Taffoni, "A formal approach to support interoperability in scientific meta-workflows," *Journal of Grid Computing*, vol. 14, pp. 655–671, Dec 2016.

[10] "Pegasus - Workflow Management System." `https://pegasus.isi.edu/`.

[11] "MinIO - High Performance, Kubernetes Native Object Storage." `https://min.io/`.

[12] F. Tessier, M. Martinasso, M. Chesi, M. Klein, and M. Gila, "Dynamic provisioning of storage resources: A case study with burst buffers," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, (Los Alamitos, CA, USA), pp. 1027–1035, IEEE Computer Society, may 2020.

[13] “Ansible - Agentless IT Automation .” https://www.ansible.com/.

[14] F. Herold, S. Breuner, and J. Heichler, “An introduction to BeeGFS,” 2014.

[15] “Cassandra - High-Performance Database.” https://cassandra.apache.org/.