HORIZON 2020
TOPIC FETHPC-02-2017
Transition to Exascale Computing

Maestro
801101

D2.3
Middleware Requirements and API Design

WP2: Middleware Requirements from Workflows

Domokos Sármány
Simon Smart
Julien Capul
François Tessier
Salem El Sayed

Date of preparation (latest version): February 2020

This project has received funding from the European Union's Horizon 2020 research and innovation program through grant agreement 801101

DOCUMENT INFORMATION

Deliverable Number	D2.3
Deliverable Name	Middleware Requirements and API Design
Due Date	February 2020 (PM 18)
Deliverable lead	ECMWF
Authors	Domokos Sármány (ECMWF)
	Simon Smart (ECMWF)
	Julien Capul (CEA)
	François Tessier (ETHZ)
	Salem El Sayed (JUELICH)
Responsible Author	Domokos Sármány (ECMWF)
	e-mail: domokos.sarmany@ecmwf.int
Keywords	[Memory Systems, Data Middleware, HPC]
WP/Task	WP2/Task(s) 2.2
Nature	R
Dissemination Level	PU
Planned Date	February 2020
Final Version Date	February 2020
Reviewed by	

DOCUMENT HISTORY

Partner	**Date**	**Comment**	**Version**
ECMWF	9/12/2019	Initial version	0.1
ECMWF	8/1/2020	Input from ECMWF, CAE, ETHZ	0.2
ECMWF	13/1/2020	Initial LaTeX version	0.3
ECMWF	14/1/2020	Input from all partners	0.4
ECMWF	22/1/2020	Maestro core API overview	0.5
ECMWF	3/2/2020	Submitted for internal review	0.6
ECMWF	19/2/2020	Review comments	1.0
ECMWF	24/2/2020	Final version for EC submission	1.1

Executive Summary

Maestro is a FETHPC-2018 funded project that will design and build a data-aware and memory-aware middleware framework for high-performance-computing (HPC) applications and workflows. In work package 2 (WP2), the partners analyse the behaviour of the applications, determine the requirements these applications impose on the Maestro system and describe how such a system can be exploited. This document builds on the earlier Deliverable 2.1 [1], which provided usage scenarios, uses cases and an initial list of requirements from all application partners. It gives a summary of the current state of the Maestro core API design and updates the list of requirements based on the co-design work that resulted in that design. It also provides a brief evaluation on how the core API accommodates the applications' requirements.

Contents

Glossary

Abbreviation	Expansion
CDO	Core Data Object, fundamental data unit understood by the Maestro middleware and communicated between the Maestro middleware and its applications
CLM	Community Land Model
COSMO	A non-hydrostatic limited-area atmospheric prediction model
CSCS	Swiss National Supercomputing Centre
DFT	Density Functional Theory, a computational method for quantum-mechanical simulation
ECMWF	European Centre for Medium-Ranged Weather Forecasts
GPU	Graphics Processing Unit, a processor optimised for display functionality
HPC	High-performance computing
IFS	Integrated Forecast System
MARS	Meteorological Archival and Retrieval System, ECMWF's perpetual archive for meteorological data
MCT	Model Coupling Toolkit
MPI	Message Passing Interface
MPMD	Multiple Programs Multiple Data Streams
NWP	Numerical Weather Prediction
OASIS3-MCT	A coupler between numerical codes representing different components components of the climate system
ParFlow	An open-source, modular, parallel watershed flow model
SCRIP	Spherical Coordinate Remapping and Interpolation Package
TSMP	Terrestrial Systems Modelling Platform, TerrSysMP
WP	Work Package

1 Introduction

This document describes an updated version of the existing requirements and new requirements that have emerged from the co-design taking place as part of WP2, WP3, WP4 and WP5 since the publication of the initial description of requirements [1].

The list of requirements are grouped into four different domains, each represented by an application partner.

1. ECMWF: Numerical weather prediction (NWP) workflow based on the operational use of the Integrated Forecast System (IFS)
2. CEA: Computational fluid dynamics and in-situ analysis
3. JUELICH: Earth modelling system TerrSysMP
4. ETHZ: Electronic structure calculation using the SIRIUS library

To evaluate the extent to which the requirements are satisfied, the current Maestro API design is summarised. Each application partner provides an updated set of requirements, including those that have been kept unchanged, modified, dropped or added anew.

2 Overview of the current Maestro architecture

The evolution of the Maestro middleware design has been documented in a series deliverables of WP3, WP4 and WP5. This section's brief overview is based on the submitted deliverables D3.1 [2], D5.2 [3], D5.3 [4] and a draft architecture document [5] that will accompany the full core middleware release (D3.3). Here we summarise the aspects most relevant to the requirements descriptions.

The Maestro project is building a data-aware and memory-aware middleware framework for HPC applications and workflows. The middleware framework can be viewed as a service aimed primarily at applications that create and use data objects. These applications may hand over control of objects and resources to the middleware and request control of data objects from it. It may also require the Maestro middleware to execute certain transformations on a data object. The Maestro middleware thus stores, moves, transforms and describes data object for the applications.

A Maestro-enabled workflow is one that consists of the following components.

- **Applications using the Maestro core API** An application participate in a Maestro-enabled workflow, and therefore become a Maestro-enabled application, by calling `mstro_initialize()` on the Maestro core API. Calling `mstro_finalize()` will stop the participation. While part of a Maestro-enabled workflow, an application would be able to use the functionality exposed by the Maestro core API. Most requirements defined in this document are imposed on this API and its behaviour.
- **Pool** Maestro-enabled applications make resources, such as memory and storage, available to the Maestro core. These resources form a pool together and are managed by the Maestro middleware.

- **Pool manager** The pool manager is an abstract entity defined by a pool-management protocol. It is responsible for managing information about a global pool of resources and their availability to applications. A global pool is defined as the set of all local pools. A local pool is directly accessible to a given process inside a Maestro-enabled application. Every task in a Maestro-enabled workflow has a local pool.

- **Workflow manager** A workflow manager is conceptually external to the set of Maestro-enabled applications, though it influences and may even control their behaviour. It coordinates application start-up and shut-down and may communicate with the pool manager directly. WP4 aims to select a workflow manager best suited for use together with the Maestro middleware. Some applications, however, may already have their domain-specific workflow managers and thus may prefer to integrate the Maestro functionality into them. This should also be possible.

- **Pre-configured standard applications** These are optional but would typically include loggers, auditors, analysis tools, telemetry tools, etc.

To evaluate how the current Maestro design matches up with the requirements, we summarise below the current designs of the Maestro core API and the pool manager.

2.1 Overview of the Maestro core middleware

The Maestro project presents a middleware framework for a diverse set of user applications with a range of requirements covering what can broadly be categorised as memory and data-awareness. Using data-aware and memory-aware abstractions, Maestro provides features for three distinct usage models. These correspond to the following three operating modes of Maestro.

1. A middleware service to support simple workflow management and coupling data between applications. The application will hand over data to the middleware and expect them to be managed by the middleware.

2. A library-only mode with a small set of available services. This will allow Maestro to perform layout-sensitive data-centric optimisations without the setup, configuration and other overheads of the management framework.

3. A hybrid mode where the full set of management and transformation features are available, although the Maestro core will put certain restrictions in place.

2.1.1 Abstractions

Figure 1 shows a small set of abstractions that the Maestro project has introduced for the core middleware.

1. *Context object* The context object captures high-level classifiers to indicate which mode the application is likely to require. Different context objects are specified for the ‘transformation’, ‘management’ and ‘hybrid’ modes.

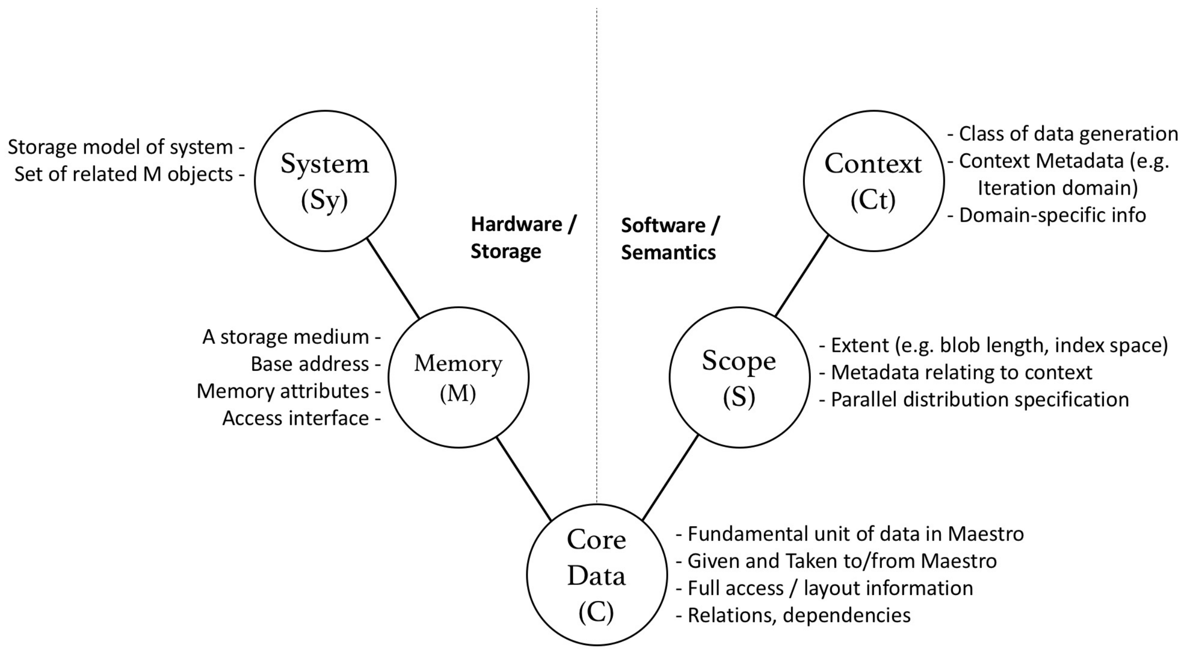
Storage model of system -
Set of related M objects -
System (Sy)
Hardware / Storage
Software / Semantics
Context (Ct)
- Class of data generation
- Context Metadata (e.g. Iteration domain)
- Domain-specific info
A storage medium -
Base address -
Memory attributes -
Access interface -
Memory (M)
Scope (S)
- Extent (e.g. blob length, index space)
- Metadata relating to context
- Parallel distribution specification
Core Data (C)
- Fundamental unit of data in Maestro
- Given and Taken to/from Maestro
- Full access / layout information
- Relations, dependencies

Figure 1: Fundamental abstractions and their relations (taken from D3.1)

2. *Scope object* The scope object captures available information about the scope, size, indexing and ordering of the data. It may also capture semantic information about the application's use of data structures attached to the data.

3. *Memory object* The memory object represents a set of locations that can be used for storage, as well as an allocation process and a set of interfaces to load and store data.

4. *System object* The system object is a set of memory objects represented as a graph as, within a system, some objects may form a network and be accessed through the edge between the nodes. The system object can also be multi-level.

5. *Core data object* The core data object (CDO) is Maestro's fundamental data type. It typically consists of a Scope Object and a Memory Object and, if so, it represents real data and their physical location. It combines all available information about the storage and the semantics and thus has the complete understanding about a particular object. A CDO's two most important qualities are its access status and pooled status: the access status determines whether its accessible via the Maestro middleware, while its pooled status determines whether the application relinquished control over the object, which then becomes part of the Maestro pool.

These abstractions help to define the APIs that support the different operating modes. Figure 2 shows these modes and their associated APIs.

2.1.2 Maestro core API

The two distinct operating modes – management and transformation – require two different APIs. The third, hybrid, mode can in principle be supported by the fusion of the

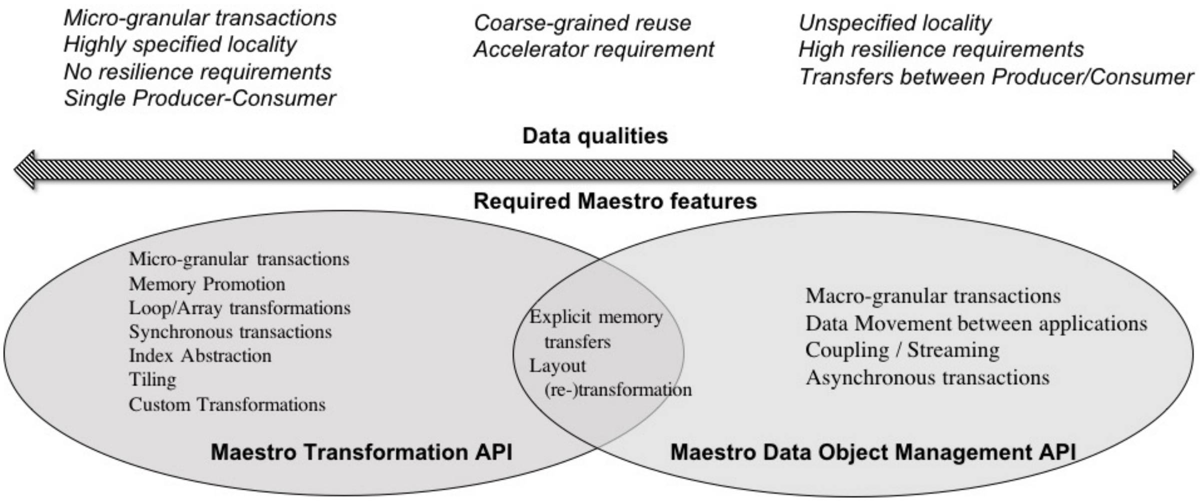
Micro-granular transactions
Highly specified locality
No resilience requirements
Single Producer-Consumer
Coarse-grained reuse
Accelerator requirement
Unspecified locality
High resilience requirements
Transfers between Producer/Consumer
Data qualities
Required Maestro features
Micro-granular transactions
Memory Promotion
Loop/Array transformations
Synchronous transactions
Index Abstraction
Tiling
Custom Transformations
Explicit memory transfers
Layout (re-)transformation
Macro-granular transactions
Data Movement between applications
Coupling / Streaming
Asynchronous transactions
Maestro Transformation API
Maestro Data Object Management API

Figure 2: The Maestro core's management and transformation operations modes and associated APIs, taken from D3.1 [2]

two APIs, although some restrictions do apply.

Management API Applications that communicate to the Maestro middleware through this API treat CDOs as a layout-independent abstract object, uniquely identified by a name. From the application's viewpoint, pooled CDOs are immutable. The Maestro middleware makes a distinction between producers and consumers, which often need to make different API calls. This does not mean that a single application cannot *act* both as a producer and a consumer, but it will have to ensure that the API calls are consistent with a given role. Both the producer and the consumer will make API calls that represent four distinct steps in the CDO's lifetime: *a)* declaration (`DECLARE`), *b)* pool injection (`OFFER/REQUIRE`), *c)* pool retraction (`WITHDRAW/RETRACT/DEMAND`), and *d)* disposal (`DISPOSE`). Table 2 summarises how a CDO may change its access and pool status throughout its lifetime according to the management protocol specified in D3.1 [2]. Note that the protocol's design is not yet finalised and there is substantial ongoing work on how to extend this protocol to satisfy more of the applications' requirements. In particular, there is a work-in-progress extension of the management API with the concept of CDO declaration `SEAL`. It provides a mechanism for early attribute communication with the aim of allowing consumers to make queries about CDOs that are not yet ready, but will be. The `DECLARE` step is thus optionally a multistep process ended by a 'seal CDO declaration' step.

Transformation API For applications that require layout transformations, the Maestro middleware provides an API that applies to mutable data. This API does not return new CDOs as outputs and thus only operates on CDOs that are accessible but not pooled (i.e. the access status is `true` and the pool status is `false`). The transformations are grouped into loop-nest transformations – such as permutation, loop splitting, loop fusing, explicit tiling and rescheduling – and layout transformations, such as transposition, index splitting, alignment, padding, reshaping and redistribution. These are defined and discussed in more detail in [2].

	Producer			Consumer		
	Protocol	Access	Pooled	Protocol	Access	Pooled
Declaration	`DECLARE`	T	F	`DECLARE`	F	F
Pool injection	`OFFER`	T	T	`REQUIRE`	F	T
Pool retraction	`WITHDRAW`	F	T	`RETRACT/DEMAND`	F/T	F/F
Disposal	`DISPOSE`	n/a	n/a	`DISPOSE`	n/a	n/a

Table 2: CDO access and pool status as specified in D3.1 [2]

Hybrid API A subset of the layout transformations above will also be made available as pooled transformation. The hybrid API will thus offer a combination of the two operation modes for the applications that find use for both the management and transformation functionality.

2.1.3 Overview of the pool manager

The concept of a pool manager emerged in D3.2. At the highest level, it is a logical entity that coordinates multiple applications into a set of cooperating Maestro-enabled applications. It manages information about

- applications using the (global) Maestro pool,
- resources available to the workflow, and
- existing CDO declarations and locations.

It is typically implemented as a distinct application. It may, alternatively, be a distributed entity formed by all those applications joining the workflow that are ready to perform pool-management functionality. (In the distributed case, one instance of the pool manager would be talking to a local group of applications and the various pool managers talking to each other.) In principle, any of the Maestro-enabled applications can act as a pool manager: the `maestro-core` library will expose a function to start the pool manager.

Access and communication Other Maestro-enabled applications access the pool manager through the Maestro core API only. This is to allow the user to only need to be aware of an abstract pool and to leave the Maestro core the one component that is required to 'speak' pool protocol. If, for example, the Maestro core instance finds a `DEMAND`ed CDO in the local pool, it will not contact the pool manager but rather retrieve the CDO directly. As a result, the pool manager is not required for single-process use because the Maestro pool will always be local to the process. If the CDO is not in the local pool, however, the Maestro core instance is responsible for forwarding application requests to the pool manager, whose role is then to locate the CDO. The pool manager is also allowed to move CDOs anywhere anytime once they are `OFFER`ed.

Data orchestration The pool manager, therefore, is the central authority for coordinating CDO movement. It knows all CDO declarations and their attributes, including where data physically is located. A pool-management protocol facilitates this coordination and all Maestro-enabled applications with pool-management responsibility must 'speak' it. All other Maestro-enabled applications communicate to the pool manager via

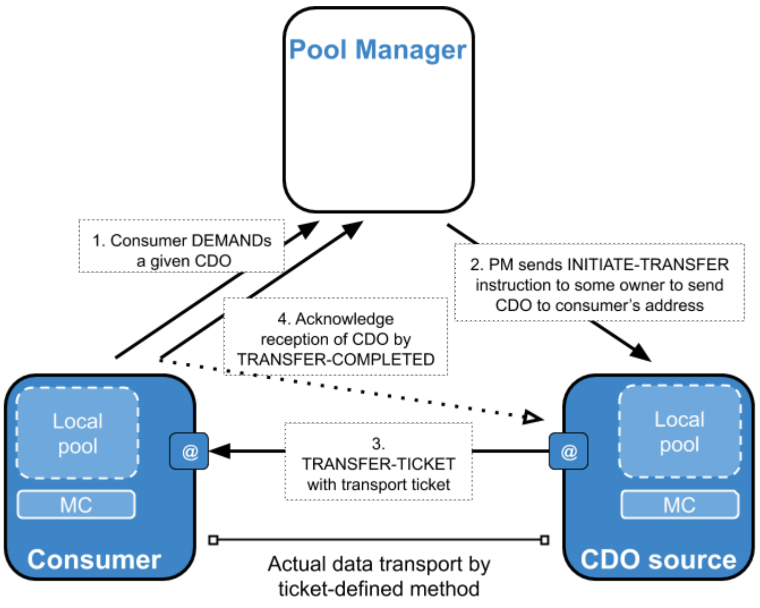
Pool Manager
1. Consumer DEMANDs a given CDO
2. PM sends INITIATE-TRANSFER instruction to some owner to send CDO to consumer's address
4. Acknowledge reception of CDO by TRANSFER-COMPLETED
3. TRANSFER-TICKET with transport ticket
Local pool
MC
@
Consumer
CDO source
Actual data transport by ticket-defined method

Figure 3: The CDO transfer protocol, taken from [5]

a Maestro-core instance. This centralised communication allows the pool manager to orchestrate CDO movement between Maestro-enabled applications because it possesses both the knowledge of the potential traffic from the application requests and the knowledge of the physical resources. The pool manager is only responsible for matching a `DEMAND` to an `OFFER`; the actual CDO transport is carried out be an adaptive transport layer, described in detail in D5.3 [4].

Scheduling From the scheduling point of view, the pool manager is responsible for low-level data-movement scheduling and data placement. This contrasts with the high-level job scheduling carried out by the workflow manager. The pool manager needs to be running before any other application enters the Maestro workflow and needs to stop after all other applications have left the Maestro workflow. It will also react to checkpoint notices from the workflow manager by notifying all components of a pending workflow preemption, then will wait for applications to complete persisting CDOs, and will finally terminate in a resumable state. Upon restarting from checkpoint data, it will ensure the persisted CDOs are visible in the (global) Maestro pool as before.

Protocol Figure 3 from [5] describes a four-step protocol for CDO transfers between the Maestro pool manager and other Maestro-enabled applications.

1. A Maestro-enabled application notices that a `DEMAND` cannot be fulfilled from local data, so it sends a request to the pool manager.

2. The pool manager matches the request with a suitable instance of the CDO data. It sends a `INITIATE_TRANSFER` request to the owner of the requested CDO, indicating

the original requestor and necessary layout attributes of the requested CDO.

3. The selected application sends a `DEMAND_ACK` to the original requestor, passing its own CDO layout attributes (this message also contains a transport descriptor). At the same time it invokes the `mstro_transport_execute()` function.

4. The original requestor invokes the `mstro_transport_execute()` function with the transport descriptor received. When this is completed it notifies the pool manager with a `TRANSFER_COMPLETED` message. At that time the pool manager knows that a duplicate copy of the CDO exists and can use this information to satisfy other requests.

2.2 Overview of workflow manager and execution framework

While the pool manager performs low-level scheduling of data movement and placement, the workflow manager is responsible for high-level scheduling of tasks, which includes job scheduling and placement. As a result, the workflow manager needs to initiate all workflow tasks, including Maestro components. The execution of a workflow would first need to start the pool manager or an application that would act as the pool manager. Next the workflow manager needs to obtain the pool manager configuration information, which is then injected into all subsequent workflow components that are part of the described workflow. During runtime, the workflow manager would observe the pool events and workflow-component status.

Given the number of available workflow managers, the operation of embedding Maestro into a workflow description needs to be extendable. Figure 4 shows the planned components of the execution environment. The aim is to inject extensions and Maestro-required components into an existing workflow description using a translator. This allows the workflow manager to execute a Maestro-enabled workflow with minimum change to the workflow description and the workflow manager itself. Specifically, for a given workflow manager to support a Maestro-enabled workflow, the translator and the workflow extensions have to be adjusted.

A study of the exiting workflow managers by WP4 has led to the selection of Pegasus, which is a widely used workflow manager in the HPC community. Pegasus provides properties that facilitate using it as a proof of concept on how to support Maestro-enabled workflows.

However, none of the use cases in WP2 uses Pegasus as workflow manager. We have therefore created a mock-up workflow to test the designed execution framework and its related components. The co-design process with WP4, including the updated requirements listed here and in [1] have inspired both the design of the execution framework and the mock-up workflow used as a demonstration. The feasibility of porting at least one of the usage scenarios to Pegasus as a means to demonstrate the execution framework depends on the efforts required to either adjust a given usage scenario to Pegasus or to create the required translator and extensions for a currently used workflow manager.

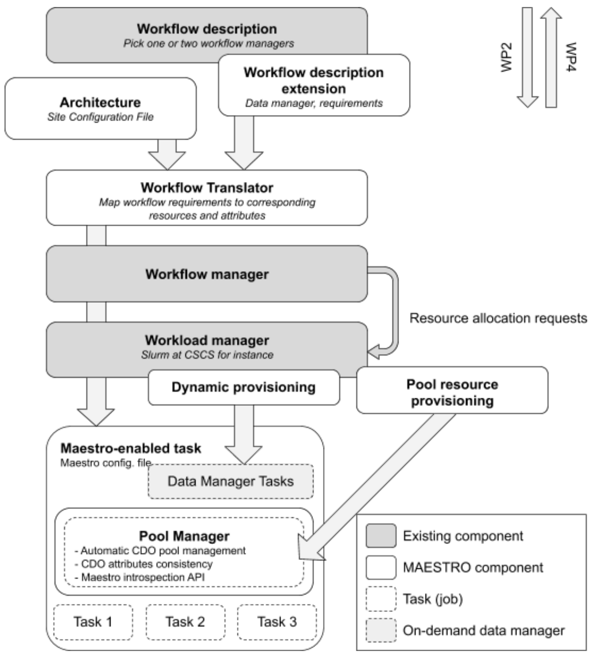
Workflow description
Pick one or two workflow managers
Workflow description extension
Data manager, requirements
Architecture
Site Configuration File
WP2
WP4
Workflow Translator
Map workflow requirements to corresponding resources and attributes
Workflow manager
Resource allocation requests
Workload manager
Slurm at CSCS for instance
Dynamic provisioning
Pool resource provisioning
Maestro-enabled task
Maestro config. file
Data Manager Tasks
Pool Manager
- Automatic CDO pool management
- CDO attributes consistency
- Maestro introspection API
Task 1
Task 2
Task 3
Existing component
MAESTRO component
Task (job)
On-demand data manager

Figure 4: Workflow execution using a workflow manager, taken from [5]

3 IFS numerical weather prediction system (ECMWF)

Deliverable D2.1 [1] identified two usage scenarios for numerical weather prediction at ECMWF: one for the time-critical operational workflow and another for research workflows, which are not time critical but exhibit unpredictable access patterns. Seven use cases derive from these usage scenarios. These remain unchanged from their original descriptions, but we list them here for completeness and to aid readability for the requirements tables that follow.

UC1.1 Access semantically-related datasets.

UC1.2 High-velocity production of meteorological objects

UC1.3 Operators restart forecast from previously computed forecast data

UC1.4 Operators monitor system characteristics

UC1.5 Operators monitor workflows

UC1.6 Sustained high-volume production of meteorological objects

UC1.7 Consumer applications request sets of objects

3.1 Unchanged requirements

Requirements that are unchanged from D2.1 [1] are listed here.

Number	**R1.6**

Desirability	Optional
Use case (in D2.1)	**UC1.1**
Description	Support describing a set of fields as ‘complete’.
Justification	Some consumers, in particular product generation, may want to iterate over ranges of data. When iterating over a sparse set of data that has been retrieved, it is important to know when the generation of all the data has been completed. This functionality could be implemented at the application level.

Number	**R1.8**
Desirability	Must have
Use case (in D2.1)	**UC1.2, UC1.6, UC1.7**
Description	Objects may have a minimum lifetime (possibly zero).
Justification	An unknown number of consumers may access a given field or set of fields, in a manner that is not known prior to runtime. Further, the operators may intervene and cause already completed tasks to run again. Maestro therefore must guarantee the existence of a field for a user-defined period.

Number	**R1.9**
Desirability	Optional
Use case (in D2.1)	**UC1.2, UC1.6, UC1.7**
Description	Objects may have a maximum lifetime (possibly infinite).
Justification	It is possible that an object is produced but no consumer ever requests it.

Number	**R1.10**
Desirability	Must have
Use case (in D2.1)	**UC1.3, UC1.4, UC1.5**
Description	When an error occurs in the Maestro software/system that impacts the ability of the Maestro system to correctly satisfy requests, it should be treated as a hard failure and be propagated to all other Maestro-dependent instances.
Justification	ECMWF’s operational system does not require best-effort to keep going at all costs. It is better to fail fast in a way that it is visible to the operators, and can be investigated by analysts quickly, than it is to stall by automatically retrying.

Number	**R1.11**
Desirability	Optional
Use case (in D2.1)	**UC1.4, UC1.7**
Description	Maestro may have ‘soft’ failure modes, if these do not impact the ability to access data according to the specification. As an example, if a duplicate copy of an object is lost, this may impact performance and may violate resiliency guarantees, but it is not a hard error from the perspective of the workflow.
Justification	Not all errors impact the external behavioural correctness of the system. Although they may have performance impacts, it is not necessary to hard-stop the entire system in this context – but it may be of use to report these errors. This is especially true if the error reports can be brought to the immediate attention of the operators. We anticipate that the implementation will default to always reporting hard errors, and softer failure modes will be added later in the development process.

Number	**R1.12**
Desirability	Should have
Use case (in D2.1)	**UC1.4, UC1.5**
Description	Errors propagate from Maestro to dependent workflow components when they next access the Maestro API.
Justification	The Maestro middleware failing hard means that it should make no best-attempt to keep going, rather than that it needs to be aggressive in tearing the system down. This makes it unnecessary to build a low-latency push system to actively propagate errors.

Number	**R1.14**
Desirability	Should have
Use case (in D2.1)	**UC1.4**
Description	The Maestro middleware should record and communicate usage statistics – type of storage used and their load, write (production) rate, read (consumption) rate, etc. – to human operators or system monitoring/logging tools.
Justification	Operators can carry out this task with ECMWF’s current system. Recorded telemetry information is extremely useful for diagnosing issues.

Number	**R1.18**
Desirability	Should have
Use case (in D2.1)	**UC1.2, UC1.5**
Description	The way Maestro handles objects in case of errors within Maestro should be well-defined and consistent.
Justification	Software and processes need to be designed around the semantics of the system. The processes for dealing with errors may be used rarely, but they must be robust, and reliable.

Number	**R1.19**
Desirability	Must have
Use case (in D2.1)	**UC1.1**
Description	Objects within Maestro are handled consistently. No risk of undefined behaviour or data corruption.
Justification	Meteorological data must only be changed according to a predefined set of rules.

Number	**R1.21**
Desirability	Should have
Use case (in D2.1)	**UC1.7**
Description	A single consumer must be able to iterate over a set of objects, even when this is too large to fit in memory at once.
Justification	If a set of objects is retrieved, and iterated over, it is desired to give the entire request to Maestro to handle in the most efficient manner possible. Maestro should provide data as fast as it can, but throttled according to the resource constraints on the consumer.

Number	**R1.23**
Desirability	Should have
Use case (in D2.1)	**UC1.7**
Description	Once consumers have obtained a data set and started iterating over the elements, Maestro must manage its resources so that it will not be out-of-memory-killed at least until the iteration completes.

Justification	It is the application's responsibility to ensure that it does not use too much memory. However, if the application is iterating over a large data set, it is not able to directly constrain Maestro's total resource usage. Maestro needs to throttle its retrieves according to resource availability on the consumer nodes.

Number	**R1.24**
Desirability	Must have
Use case (in D2.1)	**UC1.1**
Description	Once data objects are handed over to Maestro, they must be immutable.
Justification	Although data objects may have partial state during creation, once finalised they represent meteorological data that is unique. Access to this data must be consistent, and its appearance must be transactional. Any attempt to overwrite said data should produce a new (also immutable) object rather than modifying the original.

Number	**R1.25**
Desirability	Optional
Use case (in D2.1)	**UC1.2, UC1.7**
Description	Maestro may support iteration granularity larger than a single data object.
Justification	Within ECMWF's workflow, the granularity of the data required for processing is not necessarily a single field. The simplest example is that of wind data, where two fields are required. Another example would be time-series data. Multiple data elements must thus be provided together during iteration.

Number	**R1.26**
Desirability	Must have
Use case (in D2.1)	**UC1.7**
Description	Consumer applications should not be required to know the size of the data prior to requesting it.
Justification	That would be an unnecessary constraint on the application. Maestro should be able to manage memory resources or supply the required tools to enable the application to do so at the appropriate moment.

3.2 Discarded requirements

The following requirements have been dropped as part of the current review.

Number	**R1.4**
Desirability	Optional
Use case (in D2.1)	**UC1.1, UC1.7**
Description	Object retrieval may support metadata queries that describe ranges of values, for example by automatically expanding it to a list.
Justification	Consumer operations retrieve data according to the MARS language which supports such ranges. Nevertheless, we drop this requirement because the expansion could be carried out easily on the client side if required so long as R1.3 is satisfied.

Number	**R1.20**
Desirability	Should have
Use case (in D2.1)	**UC1.7**
Description	If a set of objects is being iterated over by a set of consumers, and one of these consumers fails, the entire iteration should be considered to have failed.
Justification	If a set of consumers is iterating over data in a distributed fashion, as determined by the middleware, then from an external perspective if one of the consumers fails then the entire process has failed. It is likely that the entire process will need to be rerun. However, we no longer view this to be the responsibility of the Maestro middleware.

Number	**R1.22**
Desirability	Should have
Use case (in D2.1)	**UC1.7**
Description	Multiple consumers should be able to iterate over a set of requested objects simultaneously, either all of them consuming all the objects or distributed amongst the consumers as they are able to consume them.
Justification	This would allow efficient task-based parallelism to be supported at the middleware level. The requirement is dropped, however, as this may best be implemented directly in the application if required.

3.3 Modified requirements

Requirements that have been modified are listed here. These also include requirements that are modified only in their justification.

Number	**R1.1**
Desirability	Must have
Use case (in D2.1)	**UC1.1**
Description	Objects handled within Maestro can be described and handled according to user-defined domain-specific metadata.
Justification	Scientific software is written by scientists who think in scientifically meaningful terms. One of the goals of a good data-handling system is to abstract the locations and storage-handling mechanisms away from the application developers and to provide mechanisms which facilitate their work. The Maestro middleware will substitute some existing functionality within ECMWF's workflows, which already operate according to scientific metadata described according to the MARS language. The actual implementation of the metadata object (if there is such object) may vary between workflow components but it should be able to carry information defined as above.

Number	**R1.2**
Desirability	Should have
Use case (in D2.1)	**UC1.1**
Description	User-defined metadata should take the form of a dictionary of key-value pairs, or equivalently as a set of arbitrarily named attributes.
Justification	Special, albeit favoured, case of R1.27. The Maestro system is going to substitute some existing functionality within ECMWF's workflow, which already operate according to scientific metadata described according to the MARS language. This structure queries in these terms. This is no longer a strict (must-have) requirement as we can use a schema to manipulate metadata to and from other forms. However, a dictionary of key-value pairs or a set of named attributes are the closest to native metadata support.

Number	**R1.3**
Desirability	Must have
Use case (in D2.1)	**UC1.1, UC1.7**

Description	Object retrieval should support metadata queries that describe lists of values.
Justification	This requirement has been expanded into three sub-requirements R1.3a-d. The common justification for all three is that consumer operations retrieve and operate on sets of fields. The vast majority of MARS requests use at least one list in their metadata description.

Number	**R1.5**
Desirability	Optional
Use case (in D2.1)	**UC1.1, UC1.7**
Description	Object retrieval may support wildcards, such as 'all' and 'every'. This may be interpreted as all available data matching the request.
Justification	These types of requests occur at ECMWF when researchers request data for verification and/or analysis. The requirement is marked 'optional', however, because the expansion could be carried out on the client side if required.

Number	**R1.7**
Desirability	Must have
Use case (in D2.1)	**UC1.1**
Description	Once data objects are handed over to Maestro, managing these must no longer be the application's concern.
Justification	'Managing objects' covers multiple requirements, so this has been split up into three sub-requirements; see 1.7a-c. A common justification is that if these three are not met, responsibilities Maestro is meant to substitute within ECMWF's current workflows will have to be kept.

Number	**R1.13**
Desirability	Should have
Use case (in D2.1)	**UC1.3**
Description	Maestro may have a mechanism to enable an out-of-band workflow component to inject knowledge and data into it outside of normal process.

Justification	If a workflow has to be (re)started partway through, especially after a system failure, it is necessary to get the system into a state where it can resume from a position that is not the start of the workflow. This will likely involve seeding the workflow, and hence Maestro, with data and state information out-of-band. A typical example could be when a computational node fails during operational forecast. The forecast run needs restarting from the last available forecast data. That requires injecting the available forecast data into the workflow at a point which is not the beginning of workflow under a normal operational run.

Number	**R1.15**
Desirability	Must have
Use case (in D2.1)	**UC1.2**
Description	The Maestro middleware must be able to cope with at least 20,000 object creations per second per workflow. This is sustained for most of the workflow duration.
Justification	This requirement has been expanded into sub-requirements based on the detailed descriptions of the workflow components usage characteristics. See R1.15a-d.

Number	**R1.16**
Desirability	Must have
Use case (in D2.1)	**UC1.6**
Description	The Maestro middleware must be able to support multiple non-time-critical workflows, creating at least a combined 150TiB data in an hour.
Justification	This requirement captures the non-time-critical aspect of the workload characteristics typical at ECMWF, such as archiving operational data and storing and archiving output from research experiments. It has been expanded into two sub-requirements: R1.16a-b.

Number	**R1.17**
Desirability	Must have
Use case (in D2.1)	**UC1.2, UC1.6**
Description	Object visibility within Maestro is handled transactionally. No partial state must be accessible.

Justification	The reliability and consistency of the operational workflow is paramount. Data corruption must be avoided under all circumstances, even if this reduces efficiency. As a result, data-write and indexing must be atomic and consistent from the perspective of a reading process. Either an object is not available, or the entire correct object must be returned. This is especially important if workflow components are rerun, when either old or new data must be returned (and correctly identified), but it must be impossible to retrieve part-new and part-old data in a request.

3.4 New requirements

The following requirements have been newly added as part of the current deliverable.

Number	**R1.3a**
Desirability	Must have
Use case (in D2.1)	**UC1.1, UC1.7**
Description	Object retrieval must support retrieving all objects of a list of metadata values as a single operation.
Justification	See R1.3 for a common justification.

Number	**R1.3b**
Desirability	Must have
Use case (in D2.1)	**UC1.1, UC1.7**
Description	Object retrieval must support retrieving all existing objects of a list of metadata values even if not all requested objects are available.
Justification	See R1.3 for a common justification.

Number	**R1.3c**
Desirability	Must have
Use case (in D2.1)	**UC1.1, UC1.7**
Description	Object retrieval must support querying which of a list of metadata values are currently available for retrieval.
Justification	See R1.3 for a common justification.

Number	**R1.3d**
Desirability	Must have
Use case (in D2.1)	**UC1.1, UC1.7**
Description	Object retrieval must support querying which of a list of metadata values the Maestro middleware expects to be available in the future, even if they are not currently available.
Justification	See R1.3 for a common justification.

Number	**R1.7a**
Desirability	Must have
Use case (in D2.1)	**UC1.7**
Description	Once data objects are handed over to Maestro, transport and access must no longer be the application's concern.
Justification	See R1.7 for a common justification.

Number	**R1.7b**
Desirability	Must have
Use case (in D2.1)	**UC1.1**
Description	Once data objects are handed over to the Maestro middleware, it should ensure that the data objects remain uniquely identifiable.
Justification	See R1.7 for a common justification.

Number	**R1.7c**
Desirability	Must have
Use case (in D2.1)	**UC1.1**
Description	Once data objects are handed over to the Maestro middleware, it should facilitate locating and accessing objects according their metadata.
Justification	See R1.7 for a common justification.

Number	**R1.15a**
Desirability	Must have
Use case (in D2.1)	**UC1.2**

Description	The Maestro middleware must be able to cope with around four hundred producers declaring and writing a total of at least 20,000 objects per second. The Maestro middleware must also be able to sustain that rate for the duration of a time-critical operational run, which is around one hour.
Justification	The current time-critical operational workflow at ECMWF writes up to 10 million fields, with each field being up to 20MiB in size. The write rate peaks at around 10-15000 fields per second.

Number	**R1.15b**
Desirability	Must have
Use case (in D2.1)	**UC1.2**
Description	The Maestro middleware must be able to cope with each producer running in parallel on multiple nodes and without explicit synchronisation between the producers. The producers mostly run independently from each other.
Justification	The current operational workflow at ECMWF uses 336 dedicated I/O nodes that together write up to 10 million fields in a time-critical window. The objects are written to a globally-visible Lustre filesystem.

Number	**R1.15c**
Desirability	Must have
Use case (in D2.1)	**UC1.2**
Description	The Maestro middleware must be able to cope with consumers making time-critical read operations for up to 70% of the written time-critical data.
Justification	Products that are disseminated to member states and commercial customers are also in the time-critical pipeline. Any given request to the Maestro middleware may entail multiple read operations, so the total number of requests is lower than the total number of read operations.

Number	**R1.15d**
Desirability	Must have
Use case (in D2.1)	**UC1.2**

Description	The Maestro middleware must be able to cope with multiple consumers running in parallel on multiple nodes without need for synchronisation. The consumers will also run independently from each other.
Justification	There are currently 150 dedicated nodes that run product generation, which read from the same Lustre filesystem and generate the products to be disseminated to ECMWF's member states and commercial customers. The entire workflow does and must be able to run in parallel. See R1.28.

Number	**R1.16a**
Desirability	Must have
Use case (in D2.1)	**UC1.6**
Description	The Maestro middleware must be able to support non-time-critical consumption of around 80% of the data produced by time-critical operations.
Justification	ECMWF archives around 80% of its operational forecast data. The consumer requests for this are not time-critical and may occur days after the operational run.

Number	**R1.16b**
Desirability	Must have
Use case (in D2.1)	**UC1.6**
Description	The Maestro middleware must be able to support multiple workflows running and producing objects simultaneously.
Justification	Research workflows, though numerous, are not time-critical and output from them is currently written to a different filesystem.

Number	**R1.27**
Desirability	Must have
Use case (in D2.1)	**UC1.1, UC1.2**

Description	Objects handled within the Maestro middleware can be described by three sets of attributes: • one specifying the properties of the run (e.g. operational or research, starting date/time), • one describing the data-collocation policy (typically the horizontal and vertical representation of the field), • and one defining the specific properties of the field (i.e. which parameter at what output step and vertical level). These three together could be made equivalent to describing objects according to user-defined metadata, although it is a much less desirable solution than R1.2.
Justification	The Maestro system is going to substitute some existing functionality within ECMWF's workflow that already operate according to scientific metadata. These sets of attributes are the minimally required to describe data in a scientifically meaningful way.

Number	**R1.28**
Desirability	Must have
Use case (in D2.1)	**UC1.1, UC1.2**
Description	Maestro must be able to cope with multiple producers and multiple consumers accessing objects simultaneously and in a time-critical fashion.
Justification	The time-critical nature of operational forecasting requires that producers and consumers be able run concurrently. In other words, it is not possible to wait until all producers complete their tasks before the consumers start requesting data objects.

Number	**R1.29**
Desirability	Should have
Use case (in D2.1)	**UC1.1, UC1.2**
Description	The Maestro middleware should be able to distribute/place objects among nodes according to hints supplied by third-party components.
Justification	Post-processing workloads are distributed by a broker, which can suggest probable locations for the processing. Although precise requests would not be made until later, consumers ('workers') on each of those nodes can then effectively retrieve objects as they are required.

3.5 Accommodation of requirements

Most of the above requirements are imposed on the Maestro core middleware and thus relate to the design work in WP3. Some requirements need to be met by the respective designs of the Maestro workflow manager (WP4) and Maestro's telemetry functionality (WP5). Here we provide a brief summary as to what extent those work packages' current states meet the above requirements.

Requirements on the core middleware API can in turn be grouped into basic functionality, failure resilience, queries, stress conditions, lifetime management and data grouping.

- Basic functionality, such as R1.7 and R1.24 are supported in the initial core middleware design document, Deliverable 3.1. In particular, the design of the core data object (CDO) and its 'management API' in sections 4.2.5 and 4.3.
- Fundamental (must-have) failure-resilience requirements (R1.17 and R1.19) are also covered by the CDO design in D3.1. Others – such as R1.10, R1.11, R1.12 and R1.18 – will require the design of a 'pool manager' for which work in WP3 is under way. As for R1.20, multi-consumer iteration are not supported in the current design and this requirement has been dropped.
- The stress conditions (R1.15, R1.16 and R1.28) will be evaluated by the demonstrators as part of WP6. Nevertheless, the introduction of the pool manager in the core middleware design and the four-step protocol for the CDO transfer from producer to consumer make satisfying these requirements feasible. It is still crucial that the application demonstrators provide relatively early feedback on this to WP3 to leave room for changes in the core middleware's implementation if need be.
- Most query requirements, such as R1.1, R1.2, R1.3, R1.5, and R1.27 are not covered in D3.1 [2] but work in WP3 is ongoing [5] on supporting these. The updated Maestro core API [5], in particular, allows setting arbitrary-named attributes. The query requirement R1.26 is met and is already implemented in the software release D3.2.
- Requirements on lifetime management (R1.8 and R1.9) are met as described in the corresponding sections (mainly section 4.4.2) in the core middleware API document D3.1.
- Operations on sets of CDOs (or CDO groups) are either planned (R1.21 and R1.22) or are unclear whether they will be incorporated (R1.23 and R1.25). This is currently ongoing work between the project partners. R1.6 loosely belongs to this category and the current support involves describing the 'complete set' as CDO group.
- The newly added scheduling requirement R1.29 is likely to be met with the inclusion of the pool manager and the CDO transfer in the Maestro core design.

WP4 has evaluated many existing workflow managers (including Pegasus and Kepler) and has selected Pegasus. In its existing form, it will not able to meet the workflow requirements of ECMWF's either usage scenarios. In particular, R1.28 states that the simultaneous read and write operations in ECMWF's workflows are run asynchronously and on multiple (sometimes hundreds of) nodes. Also, requirement R1.13 posits that the workflow should be able to run out-of-band to facilitate restarting from checkpoint data. However, these shortcomings need not mean that ECMWF cannot run a Maestro-enabled workflow. But it does suggest that ECMWF will not be able to use the Maestro workflow functionality or will need to reimplement it. WP4 efforts target alleviating Pegasus's shortcomings in addition to simplifying extending the execution framework to support other workflow managers.

The extent to which the telemetry requirement R1.14 will be met is unclear at this stage. A high-level design of the telemetry infrastructure has been provided [6]. It proposes focusing on post mortem telemetry analysis, rather than runtime analysis, within the Maestro project. Detailed specifications or a working prototype are yet to be provided.

4 Computational fluid dynamics and in-situ analysis (CEA)

The CEA workflow provided for Maestro is representative of a typical chaining scenario of two simulation codes plus two post-processing analyses of the second simulation output data. It is based on the open source proxy application Hydro, a 2D hydrodynamic simulation code, to model the two simulation codes. The two post-processing applications are a custom python-based analytical processing and a Paraview-based graphical processing. The workflow applications used CEA's Hercule I/O library and its associated file format to exchange data.

The Maestro-enabled version will consist in modifying the Hercule I/O library to allow using Maestro CDOs to produce and/or consume simulation data with the primary objective of executing the two processing steps, no longer post-, but in situ in a loosely coupled fashion (also called in transit). In other words, it consists in executing in parallel the second Hydro simulation and the two processing applications, with the simulation output data 'streamed' from the simulation to the two processing applications.

Five use cases derive from the CEA workflow. These remain unchanged from their original descriptions, but we list them here for completeness and to aid readability for the requirements tables that follow.

UC2.1 Applications write/read Data Collections to/from Maestro

UC2.2 'Streaming' data records between producers and consumers

UC2.3 Ownership and persistence of a data

UC2.4 Simulation checkpoints spooling

UC2.5 Time step rollback on persisted data

4.1 Unchanged requirements

Requirements that are unchanged from D2.1 are listed here.

Number	**R2.1**
Desirability	Must have
Use case (in D2.1)	**UC2.1**
Description	Maestro must allow a conceptual data organization based on the following data container concepts: • array: a multi-dimensional typed array • record: a grouping of related datasets published at the same time (e.g. simulation output fields at a specific time step) • collection: a grouping of related records (e.g. checkpoint data collection, post-processing data collection)
Justification	Most simulation codes and tools at CEA use the in-house Hercule library to perform IO related tasks. To minimize effort to use Maestro, we intend to port Hercule on top of Maestro (using Maestro as a backend store). This is a fundamental organization of data within Hercule.

Number	**R2.3**
Desirability	Should have
Use case (in D2.1)	**UC2.1**
Description	Maestro should provide a way to hierarchically group related data arrays within a record (and query this group structure), similar to HDF5 groups.
Justification	This will assist in porting the Hercule I/O library on top of Maestro (though this hierarchical organization could be implemented in the data naming, having a native system for hierarchical grouping would be more efficient).

Number	**R2.5**
Desirability	Should have
Use case (in D2.1)	**UC2.2**
Description	In ‘streaming’ mode, a data collection should accommodate multiple consumers organized into groups, similar to Kafka groups (or Redis Stream consumer groups). Records are broadcasted to all consumer groups. Consumers within a consumer group retrieve records from a queue.

Justification This allows mixing two types of pattern:

- all records: consumers registering in different groups receive (and process) all the records, e.g. a statistical processing that integrates data over time needs all records
- fair-share: consumers registering in the same group split the work by concurrently consuming records from a queue (hence a record will be processed by only one consumer within the group), e.g. a stateless processing such as an image rendering can work independently on a record and does not need the whole sequence

---

Number **R2.6**

Desirability Should have

Use case (in D2.1) **UC2.2, UC2.3, UC2.4**

Description Maestro should provide a configuration system to describe properties of the workflow/pipelines for which it will manage the data exchange, where these properties are known in advance.

Justification An example property that could be leveraged by Maestro at configuration time would be the identifiers of data that are known to be consumed by a consumer application. Let's say a simulation code dumps many physical quantities, but for a particular workflow, the consumer application only requires a few of them. This is an information that Maestro could leverage to avoid transporting data from the producer application that will not be consumed at all.

Number **R2.7**

Desirability Should have

Use case (in D2.1) **UC2.2**

Description Relates to R2.5.

In 'streaming' mode, a consumer should have the opportunity to declare in advance (in configuration) the data arrays it will consume.

Justification In CEA's simulation pipelines, upstream simulation codes produce many arrays that are not necessarily consumed by downstream consuming applications. The arrays consumed is information that is often available ahead of time (e.g. the user knows that its consuming application only reads pressure fields). Maestro could leverage this knowledge to optimise data flow (filtering) between producer and consumer applications.

Number	**R2.10**
Desirability	Should have
Use case (in D2.1)	**UC2.2**
Description	Relates to R2.8. When used for persisting data produced by an application, Maestro should provide the ability to specify a subsampling rate of persistence or filter.
Justification	This allows different branches of a pipeline to work on different data flow rates. In particular, the main simulation branch executed 'in transit' can work on a higher data flow rate to have refined data processing.

4.2 Modified requirements

Requirements that have been modified are listed here. These also include requirements that are modified only in their justification.

Number	**R2.2**
Desirability	Must have
Use case (in D2.1)	**UC2.1**
Description	Maestro must provide the ability to add (and query) named attributes (metadata) to data arrays, records and collections, where these terms are defined in R2.1.
Justification	As Hercule I/O library (which will use Maestro) manipulates data objects but also groups of data objects, the ability to use named attributes on data objects and on groups of data objects will facilitate it.

Number	**R2.4**
Desirability	Must have
Use case (in D2.1)	**UC2.2**
Description	A data collection must accommodate records produced by multiple independent producers (see R2.1 for the definition of 'collection' and 'records'). Hence, labelling of data records is at least a 2-tuple made of a sequence integer (the sequence number of a record with the list of records) and a producer ID integer. Maestro must provide an indexing or attribute system to be able to support this.

Justification	When one of the simulations writes data with Hercule I/O library, each rank produces a record which is self-describing and contains all data and metadata produced by that rank. Unlike in other I/O libraries, these records are not combined to produce a global view of the domain (or a different domain decomposition). They are written as is (keeping the same domain decomposition). The recombination into a different domain decomposition is deferred until reading if needed. This is the default behaviour and is part of optimisation for writing. This principle implies that records are indexed by their issuing sequence (here the time step number) and by the producer id (here the MPI rank). Therefore, to allow porting Hercule to leverage Maestro, it must provide an indexing or attribute system compatible with this approach.

Number	**R2.8**
Desirability	Must have
Use case (in D2.1)	**UC2.2**
Description	Maestro must provide the ability to persist indefinitely a data object provided that one of the storage tier managed by Maestro is non-volatile.
Justification	Required if Maestro is to replace an existing I/O stack.

Number	**R2.9**
Desirability	Should have
Use case (in D2.1)	**UC2.2**
Description	Relates to R2.8 When used for persisting data produced by an application, Maestro should provide the ability to configure which group of data objects is persisted.
Justification	An application may produce different data collections serving different purposes. The user should be able to instruct Maestro which collection needs to be persisted.

4.3 Discarded requirements

The following requirement has been dropped as part of the current review.

Number	**R2.11**
Desirability	Must have

Use case (in D2.1)	**UC2.3**
Description	When used for persisting checkpoint data, Maestro must provide the ability to specify the number of recent checkpoint records that will eventually be persisted.
	Discarded.
Justification	This is required for spooling the checkpoint records and saving persistent storage space. This can be managed by the Maestro backend in Hercule I/O library.

Number	**R2.12**
Desirability	Must have
Use case (in D2.1)	**UC2.5**
Description	When used for persisting data, Maestro must be able to detect rollback on a collection based on the record sequence index number and proceed with rewriting the persisted record history.
	Discarded.
Justification	This is a required behaviour by simulation codes at CEA.
	This can be managed by the Maestro backend in Hercule I/O library.

Number	**R2.13**
Desirability	Optional
Use case (in D2.1)	**UC2.5**
Description	When used for streaming data between consumers and producers, Maestro should be able to detect time step rollback from producer applications and clean up the staging/buffering area from discarded time steps that have not yet been consumed by consumer applications.
	Discarded.
Justification	This would save space in Maestro data staging/buffering area.
	This can be managed by the Maestro backend in Hercule I/O library.

4.4 Accommodation of requirements

Status below are split between requirements that are imposed on the Maestro core middleware and thus relating to the design work in WP3 and the requirements associated with functionalities provided by the higher level middleware layer providing workflow management capabilities and related to the WP4.

Requirements related to Maestro core middleware (WP3):

- Data organization requirements (R2.1, R2.2, R2.3) will be addressed through the CDO groups and namespace functionalities.
- Requirement for streaming pattern (R2.5) is not planned to be addressed as there is no concept of CDO streams in Maestro core middleware design. CDO can only be accessed through their name. This requirement was tagged as "Should have" since our workflow scenario consists in producing sequences of CDOs with predictable naming, hence consumers will be able to forge names for the sequence of CDOs they need to consume. However, CDO streams is a feature that we envision to require in the near future.
- Requirement R2.4 is related to indexing and accessing CDO which is deemed feasible through CDO namespaces.
- Workflow configuration and hints described in R2.6 and R2.7 are planned to be addressed partly in WP3 (based on Section 4 of D3.2) and WP4.
- Requirements related to persistence of data objects (R2.8, R2.9 and R2.10) may need some further discussions with respect to the planned implementation of Maestro core middleware. In D3.1, it is specified that CDO lifetime is bound to the lifetime of the Maestro workflow the CDO belongs to, but this latter concept is unclear as to what it emcompasses. Our understanding is that CDO lifetime should rather be bound to the lifetime of the Maestro server/management service as well as the lifetime of any non-volatile memory tier the Maestro service manages. R2.8 has been updated according to this understanding.

Requirements related to "rollback" (R2.11, R2.12 and R2.13) have been discarded on the basis that it can be implemented within the higher level application that will uses Maestro (Hercule I/O library).

Requirements related to Maestro workflow management layer (WP4) are those related to workflow-wide information gathering (R2.6 and R2.7). The implementation status of these requirements is pending further information regarding the design of this middleware layer.

5 Electronic Structure Calculation (ETHZ)

We identified two usage scenarios in Deliverable D2.1 [1] for the SIRIUS library (domain specific library for electronic structure codes developed at ETHZ/CSCS). Those two scenarios involve the DFT [1] Loop mini-app used to benchmark all components of the DFT self consistency cycle – diagonalisation of the Kohn-Sham Hamiltonian, charge density construction, mixing and generation of the effective potential – using a pseudopotential or full-potential method. The use-cases remain unchanged:

- **UC3.1** Move data from CPU to GPU memory
- **UC3.2** Management of memory resources

[1]Density Functional Theory, a computational quantum mechanical modelling method.

5.1 Unchanged requirements

Requirements that are unchanged from D2.1 are listed here.

Number	**R3.1**
Desirability	Must have
Use case (in D2.1)	**UC3.1, UC3.2**
Description	Grant ownership of data to Maestro
Justification	Maestro needs to be able to take ownership of data to orchestrate data movement.

Number	**R3.2**
Desirability	Must have
Use case (in D2.1)	**UC3.1, UC3.2**
Description	Indicate Maestro to move or allocate data to a particular tier of memory.
Justification	Explicit usage of GPU and Cuda kernel is very limiting. Maestro can abstract some of the details.

Number	**R3.3**
Desirability	Must have
Use case (in D2.1)	**UC3.1**
Description	Require data owned by Maestro
Justification	An application may need to use all or parts of data owned by Maestro even if it's not the original producer.

Number	**R3.4**
Desirability	Must have
Use case (in D2.1)	**UC3.1, UC3.2**
Description	Maestro is able to manage memory resources (including allocation and deallocation) across multiple layers of the memory hierarchy.
Justification	The multiplicity of memory and storage tiers, usually with a dedicated software stack, on modern HPC systems make them hard to use. A memory abstraction is necessary in Maestro for performance, usability and extendibility purposes.

Number	**R3.5**
Desirability	Optional
Use case (in D2.1)	**UC3.1**
Description	Data can be moved synchronously or asynchronously
Justification	Compute and communication should overlap for improved performance, but block where required for correctness.

Number	**R3.6**
Desirability	Must have
Use case (in D2.1)	**UC3.1**
Description	Maestro can move data from one memory space to another
Justification	Moving or copying data from one tier to another can be challenging given the different characteristics of the memory and storage technologies. An example could be to move data from a byte-addressable to a block-addressable memory.

Number	**R3.7**
Desirability	Must have
Use case (in D2.1)	**UC3.1**
Description	Maestro can manage different memory spaces within a node
Justification	As we may want to move data between two node-local memory tiers, each memory type has to expose its own memory space.

Number	**R3.9**
Desirability	Should have
Use case (in D2.1)	**UC3.2**
Description	Maestro should manage (re)use of the memory space with minimal overhead.
Justification	Allocation and deallocation of memory spaces can significantly decrease I/O performance. Memory space reuse can reduce this.

5.2 Discarded requirements

Number	**R3.8**
Desirability	Must have

Use case (in D2.1)	**UC3.1**
Description	Data dependencies are preserved within Maestro
Justification	The original justification was: "A calling process or application usually must deal with multiple chunks of a dataset that show interdependencies. A simple illustration is a set of tasks that needs to be performed sequentially. It is necessary to preserve those dependencies to keep data coherency and properly schedule data movements". We dropped this requirement as dependencies still have to be explicitly managed by the application developer. It is out of the scope of Maestro to handle tasks dependencies.

5.3 Accommodation of requirements

The aforementioned requirements are all related to core features of Maestro (WP3). However, we can distinguish two categories:

- **Memory management** requirements (R3.2, R3.4, R3.6, R3.7, R3.9) will be addressed by the memory abstraction, based on Mamba, that is under development.
- **Data management** requirements (R3.1, R3.3, R3.5) are core features of Maestro. The Maestro API as defined in D3.2 [5] already meets some of those requirements. R3.5, however, needs to be further investigated. Discussions have to be started with WP3 partners in that sense.

6 Global Earth Modelling System (Jülich)

TerrSysMP or TSMP is composed of three model components COSMO, CLM and Parflow, which are coupled using OASIS3-MCT. The main target is to simulate the interactions between lateral flow processes in river basins and lower atmospheric layers. In Maestro we will focus on providing the coupling of two models, namely CLM and Parflow. The Community Land Model or CLM models the effect of terrestrial ecosystems on climate, while PARellel FLOW or Parflow is an integrated hydrology model used to simulate surface and subsurface flow. Most requirements given below are inspired by the coupling needs of the TerrSysMP models. Further requirements rise when seeking better load balancing between the models for a given system allocation and experiment.

Two use cases derive from the TerrSysMP usage scenario. These remain unchanged from their original descriptions, but we list them here for completeness and to aid readability for the requirements tables that follow.

UC4.1 Exchange 2D fields between climate models

UC4.2 Dynamic Load Balancing of Coupled Models

6.1 Unchanged requirements

Requirements that are unchanged from D2.1 are listed here.

Number	**R4.1**
Desirability	Must have
Use case (in D2.1)	**UC4.1**
Description	Redistribution of parallel 2D fields
Justification	The 2D data has to be redistributed according to the distribution of the rows of the interpolation matrix (built from SCRIP) among processes.

Number	**R4.2**
Desirability	Should have
Use case (in D2.1)	**U4.1**
Description	Row and column-major layout transformation of distributed arrays
Justification	Climate models in TSMP are written in C as well as Fortran, arrays exchanged between them often have to be rearranged.

6.2 Modified requirements

Requirements that have been modified are listed here. These also include requirements that are modified only in their justification.

Number	**R4.4**
Desirability	Must have
Use case (in D2.1)	**UC4.2**
Description	Start and stop multiple MPMD runs within the same job allocation
Justification	The redistribution of coupled models over different processes is required to repeatedly measure performance to determine a better load balance for a given system and experiment

6.3 Discarded requirements

The following requirement has been dropped as part of the current review. This will be handled directly in the TerrSysMP models as part of the porting effort to Maestro.

Number	**R4.3**
Desirability	Optional
Use case (in D2.1)	**UC4.1**

Description	Interpolation of distributed arrays
Justification	Fields in TerrSysMP can have different grid sizes and can be distributed on different number of processes. A matrix representing the interpolation weights is either generated or read from a SCRIP file and used to map the distributed source array to the distributed target array. The SCRIP bilinear interpolation operator is used for COSMO variables and the SCRIP distance-weighted averaging operator for CLM variables. An MCT datatype describes the coupled system processor layout. MCT stores coupling field data in an object that supports arbitrary numbers of real- and integer-valued fields, indexed using string tokens. A domain decomposition descriptor (DDD) object uses a 1-D global index space (e.a. linearization) to represent multidimensional index spaces. Parallel communication schedules are computed automatically from source and destination DDDs. Parallel data transfer is accomplished by calling paired send/receive methods with data storage and communication schedule datatypes as inputs. MCT provides distributed storage for precomputed interpolation coefficients from which it derives communication schedules for parallel interpolation. This operation can optionally be constrained to give identical results on different numbers of processes.

6.4 Accommodation of requirements

The requirements imposed by TerrSysMP are supported by efforts done in Maestro core middleware (WP3) and the execution framework (WP4).

- The redistribution of parallel (R4.1) and row and column-major layout transformation of distributed arrays (R4.2) will be directly supported by the Mamba library and are core concepts of CDO handling in Maestro
- Start and stop of MPMD runs within the same job allocation (R4.4) will depend on the execution framework components developed in WP4
- Interpolation of distributed arrays (R4.3) is an optional requirement that as of now is not planned into the Maestro core middleware architecture

7 Concluding remarks

This document has finalised the requirements the applications impose on the Maestro core middleware. It has built on previous work on workload characterisation [1] that has described – from each of the four application partners – the usage scenarios and use cases relevant for the Maestro project. WP3 and WP5 developments in the Maestro core API design [2, 5, 4] had fed into establishing the extent to which the initial requirements needed to be refined or modified.

As a result, this document has also provided a status overview of the Maestro core API design. It has then grouped each application partner's set of requirements into four categories – whether a given requirement has been modified, discarded, kept unchanged or added anew. A brief evaluation of how the current Maestro middleware design meets, or is on course to meeting, the application's set of requirements is also provided.

References

[1] Domokos Sármány, Simon Smart, Teodor Nikolov, Julien Capul, Sebastien Morais, and François Tessier. *Maestro D2.1* Workload characterisation and Middleware Requirements, 2019.

[2] Christopher Haine, Utz-Uwe Haus, and Adrian Tate. *Maestro D3.1* Initial Core Middleware Specification, API Document, 2019.

[3] Sining Wu, Ganesan Umanesan, and Sai Narasimhamurthy. *Maestro D5.2* storage data mapping and memory integration design, 2019.

[4] Christopher Haine, Utz-Uwe Haus, and Adrian Tate. *Maestro D5.3* Adaptive Transport Development, 2019.

[5] Christopher Haine, Utz-Uwe Haus, and Adrian Tate. *Maestro D3.2++* Core Middleware Reference Implementation Architecture, 2020. In preparation.

[6] Nguyen Laurent, Sining Wu, Sai Narasimhamurthy, and Ganesan Umanesan. *Maestro D5.1* telemetry design, 2019.