This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 777533.

PROviding Computing solutions for ExaScale ChallengeS

D4.3	Updated requirements analysis, validation of 1st prototype, and updated PROCESS architecture		
Project:	PROCESS H2020 – 777533	**Start / Duration:**	01 November 2017 36 Months
Dissemination[1]:	Public	**Nature[2]:**	**R**
Due Date:	30.04.2019	**Work Package:**	**WP 4**
Filename[3]	PROCESS_D4.3_UpdatedRequirementsAnalysis_v1.0.docx		

ABSTRACT

After 18 months, the requirements analysis and the PROCESS architecture (elaborated during 6 months) were revised together with validation of the first PROCESS platform prototype. According to it, this deliverable is divided into four parts. The first part provides updated requirements analysis coming from all of the use cases. It is carried out in two steps. The first step evaluates requirements recognized in the deliverable *D4.1: Initial state of the art and requirements analysis, initial PROCESS architecture*. The second step recognizes the new requirements. The second part of this deliverable deals with validation of the 1st PROCESS platform prototype, which is described in the deliverable *D6.1: First prototype*. The next part of the deliverable is related to update of the PROCESS architecture. The initial version of the architecture was presented in the deliverable *D4.1: Initial state of the art and requirements analysis, initial PROCESS architecture* along with suggestions for architecture improvement. The architecture was improved accordingly. The final part of the deliverable describes the reference exascale architecture, which represents abstraction applicable to a wide range of scientific exascale systems.

[1] PU = Public; CO = Confidential, only for members of the Consortium (including the EC services).

[2] R = Report; R+O = Report plus Other. Note: all "O" deliverables must be accompanied by a deliverable report.

[3] eg DX.Y_name to the deliverable_v0xx. v1 corresponds to the final release submitted to the EC.

Deliverable Contributors:	Name	Organization	Role / Title
Deliverable Leader[4]	Hluchý, L	UISAV	Deliverable coordinator
Contributing Authors[5]	Bobák, M., Tran, V.	UISAV	Writers
	Somoskői, B.	LSY	Writer
	Heikkurinen, M., Höb, M., Schmidt, J.	LMU	Writers
	Graziani, M., Müller, H.	HES-SO	Writers
	Maassen, J., Spreeuw, H., Madougou S.	NLESC	Writers
	Belloum, A., Cushing, R.	UvA	Writers
	Meizner, J., Nowakowski, P.	AGH/AGH-UST	Writers
Reviewer(s)[6]	Dlugolinský, Štefan	UISAV	Reviewer
	Heikkurinen, Matti	LMU	Reviewer
Final review and approval	Höb, Maximilian	LMU	Reviewer

Document History

Release	Date	Reasons for Change	Status[7]	Distribution
0.0	31.01.2019	The initial structure of the document	Draft	All
0.1	07.02.2019	The final structure of the document	Draft	All
0.2	21.02.2019	Initial text for all sections	Draft	All
0.3	07.03.2019	Second update of all sections	Draft	All
0.4	11.04.2019	Finalization of reference exascale architecture	Draft	All
0.5	15.04.2019	Finalization of architecture updating	Draft	All
0.6	15.04.2019	Finalization of requirement analysis	Draft	All
0.7	16.04.2019	Review of the draft version	In Review	All
0.8	25.04.2019	Finalization of validation	In Review	All
0.9	26.04.2019	Language correction, review of the pre-final version	In Review	All
1.0	30.04.2019	Final version	Released	All

[4] Person from the lead beneficiary that is responsible for the deliverable.
[5] Person(s) from contributing partners for the deliverable.
[6] Typically, person(s) with appropriate expertise to assess the deliverable quality.
[7] Status = “Draft”; “In Review”; “Released”.

Table of Contents

Executive Summary

This deliverable contains updated requirements analysis, validation of the first prototype, and refined PROCESS architecture (including the reference exascale architecture). The update of requirements analysis was carried out in two steps. The first step evaluated requirements (presented in the deliverable *D4.1: Initial state of the art and requirements analysis, initial PROCESS architecture*) against the experience and knowledge gained in implementing the PROCESS platform components. The second step recognizes the new requirements that were identified during the implementation of the initial versions of the application prototypes. In addition to the general requirements stemming from the current visions of the emerging exascale systems, the project's use cases listed below provided specific, concrete requirements:

- UC#1: exascale computational processing methods that are able to exploit accelerated computing
- UC#2: scalable workflows that are able to cooperate with exascale datasets
- UC#3: support for the emerging user communities ("long tail of science")
- UC#4: responding to a large number of requests within milliseconds per request
- UC#5: methods suitable for exascale data extraction

The common requirement analysis summarizes these requirements and aligns them with the general requirements and challenges stemming from harnessing the emerging exascale systems for research and innovation actions in an efficient and flexible manner. This analysis forms the foundations of the reference exascale architecture and its key features (such as distributed file system spanning a large number of heterogeneous computing infrastructures, adaption of the virtualization and modular micro-service infrastructure). The activity was concluded by exploring the consequences of the updated requirements analysis on the PROCESS platform.

The PROCESS consortium presented the first prototype of the PROCESS platform after 12 months. The prototype is focused on extremely large computing infrastructure and is deployed on the HPC hardware infrastructure. The prototype is integrated with use cases and validated based on the approach presented in the deliverable *D3.1: Performance modelling and prediction*. The validation process took 6 months and was mostly performed at the Prometheus supercomputer[8].

The key conclusion of the analysis and validations steps was that the architecture described in the deliverable D4.1 needed several updates that in some cases necessitated quite fundamental changes. The resulting PROCESS architecture will be used as the foundation of the further implementation steps and proposed as a generalizable reference architecture for exascale applications.

[8] http://www.cyfronet.krakow.pl/computers/15226,artykul,prometheus.html

List of Figures

List of Tables

1 Introduction

Based on the experience gained during the first half of the project, most of the significant parts from the deliverable D4.1 were deemed to be in need of an update. However, the common requirements have remained practically unchanged, indicating that the initial assessment of the use cases was correct. The core requirements of the use cases are the following:

- UC#1: exascale computational processing methods that are able to exploit accelerated computing
- UC#2: scalable workflows that are able to cooperate with exascale datasets
- UC#3: support for the exascale community
- UC#4: responding to a large number of requests within milliseconds per request
- UC#5: methods suitable for exascale data extraction

While there were no major changes in the requirement analysis of the use cases, the adoption of the PROCESS architecture provided several new insights. The approach is based on the paradigm of combining the service-oriented architecture approach with cloud and high-performance solutions. This is done by the advanced virtualization layer.

Among the main challenges of exascale are elasticity, and processing of extremely-large datasets (often exceeding petabytes). To address this challenge, new management approaches have been investigated, leading on shifting of the focus to self-managing and provision of reliable infrastructure services through automation. A related, complex problem is increasing the application performance and scaling. The best solution of the above-mentioned challenges and issues is a platform of distributed services based on virtualization principles, which necessitates containerization of applications, refactoring software stacks using micro-service approach, and supporting orchestration of infrastructure.

Between project months 6 and 18, the possibilities of containerization were investigated. Currently, the concept was applied at two levels: (i) application prototypes and (ii) micro-infrastructure of the distributed virtual file system. Since one of the main characteristics of the PROCESS platform is modularity, the adoption of the containerization is very essential and natural. To ensure broadest applicability of the approach for diverse application and system components, the PROCESS architecture is based on the paradigm where components are connected together by REST-APIs.

The first prototype of the PROCESS platform which was released internally in the project month 12 validation step was concluded by month 18. The work documented in this deliverable is a result of common work across several work packages. WP4 is responsible for the design of novel architecture, but the scope of the deliverable is wider. According to this, the main goals were (with the responsible work package):

- updated requirements analysis (WP2)
- validation of the first prototype (WP8)
- reference exascale architecture (WP4)
- updated PROCESS architecture (WP4)

While a significant part of the deliverable refers to different deliverables, each of its part ends with a list of related deliverables to make the connections clear (references are expressed as footnotes).

2 Updated requirements analysis

This section presents the updated requirements analysis based on D4.1. In this deliverable, the initial software, hardware, security, and privacy requirements were defined for all the five use cases. After 18 months, they are now questioned critically and updated if necessary.

Within the common conceptual model, we initially derived a set of generalized requirements from previously described requirements. In section 2.7 we evaluate if these generalized requirements are still valid. Based on the common conceptual model an ensemble of building blocks formed a solid basis for the overall PROCESS architecture, which is updated in section 5.

2.1 Use Case 1: Exascale learning on medical image data

The requirements analysis in D4.1 pointed out the need for Docker containers in the software environment. Singularity containers, are generally deemed to be more suited to the High Performance Computing (HPC) clusters, as they provide essential features such as improved security and portability and the ability to restrict the permissions of a code run using the “root” user permissions. While Docker containers are more flexible and used more often in medical research applications, Singularity containers can be used to package entire scientific workflows, software, and libraries.

As a consequence of these conflicting requirements, the need of a flexible conversion from Docker containers to Singularity arises to guarantee software integration. Furthermore, access to GPUs has to be ensured even in the Singularity framework, as it is essential for UC#1. The rest of the requirement analysis is mostly confirmed by our observations, with only slight variations of the software requirements mentioned in D4.1

The software requirements listed in Table 1 of D4.1 have been updated by reducing the number of necessary technologies. These updates reflect actual implementation of the use case prototype, which has been realized by optimizing the number of necessary technologies.

Table 1: Overview of UC#1 requirements from the deliverable D4.1.

Currently used technologies:	Python 2.7, Tensorflow 1.4.0, Caffe, Theano, Lasagne, DIGITS, mxnet, Keras 2.1.2, TFLearn, Numpy, SciPy, Pandas, Scikit-learn, OpenCV, Openslide, Matplotlib, Seaborn, Skimage, h5py, PyTorch, OpenBLAS, cuDNN, FFmpeg, NLTK, Gensim, R, Weka.
Data Storage:	NAS
Data Processing	H5DS, Apache Spark, Hadoop
Existing computing infrastructure:	8 GPUs

Table 2: Update of UC#1 requirements from the deliverable D4.1.

Technologies used for 1st prototype:	Python 2.7, Tensorflow 1.6.0, Keras 2.1.2, TFLearn, Numpy, SciPy, Pandas, Scikit-learn, OpenCV, Openslide, Matplotlib, Seaborn, Skimage, h5py, PyTorch, OpenBLAS, cuDNN.

Especially, the support for Caffe, Microsoft CNKT and ASAP are no longer necessary, since similar functionality (with higher degree of efficiency) is offered by Tensorflow, Keras

and Openslide. The Tensorflow and Keras libraries, however, are constantly subject to updates. The latest releases generally correct malfunctions of the old releases. For that reason, the software requirement for Tensorflow has been changed from an older version 1.4.0 to the latest 1.6.0.

The initial experiments on the prototype have led to identification of additional requirements, such as:

1. Data transfer has to follow the SCP protocol. The hospital and institutions may not have open FTP access (typically for reasons related to security and policies used to achieve regulatory compliance), which makes the applicability of FTP transfer limited. Data transfer through the SCP protocol is also more flexible and could apply to different types of users.
2. Flexibility in switching between different Tensorflow and Keras versions as well as environments is needed in order to exploit functionality of different library updates.
3. Support for Uber's Horovod tool[9] is needed to parallelize GPU training across the network. Horovod is a distributed training framework for TensorFlow, Keras, PyTorch, and MXNet. This is particularly relevant to UC#1, which aims at distributing software originally deployed on single-GPU nodes. Moreover, Horovod achieves 90% scaling efficiency for both Inception V3 and ResNet-101, and 68% scaling efficiency for VGG-16.

Hardware requirements remain unchanged as well as requirements on Security and Privacy (since D4.1).

2.2 Use Case 2: Square Kilometre Array/LOFAR

The choice of the front-end solution for the pilot application is still open. Ideally, we would reuse the Web UI developed within the EU EOSC pilot project to provide an UI that is both user-friendly and familiar to most of the user community targeted by the UC Mechanisms for launching workflows are being included in this component. The latter can also be used to monitor the jobs. For downloading outputs, we can rely on the staging functionality of data services.

The possibility to use Singularity containers as workflow steps is still desired as it allows each step to use different analysis tools. Docker containers could also be used, but the workflow steps in question are executed on an HPC cluster nodes that typically support only Singularity for reasons discussed in the section 2.1.

Although work is being done to relax the necessity of fat nodes (see section 5.2 for details), we conservatively require their presence as of now. Above work consists in parallelizing the code which requires fat nodes, it is only at an exploratory stage and we do not have a guarantee it will work out.

For efficiently transferring astronomical data from the archival locations to the processing locations, we are considering using the so-called data transfer nodes (DTNs). DTNs are dedicated systems, purpose-built and tuned to facilitate high-speed file transfer over wide area networks. They typically run software designed for high-speed data transfers to a remote node (such as GridFTP) and are equipped with high-speed network interfaces (10-100Gbps).

The capability to horizontally scale to a significant number of computing resources in order to run in parallel, a large number (up to ~1800 assuming each workflow is running an

[9] The goal of Horovod is to make distributed Deep Learning fast and easy to use. The primary motivation for this tool lies in its deployment, which has been facilitated to transform a single-GPU TensorFlow application into a multi-GPU distributed architecture. See also: https://eng.uber.com/horovod/

entire observation and we target the entire LTA) of independent workflows is still desired. Since processing the archive for a single science case already requires a significant amount of core-hours O(47M), handling multiple science cases simultaneously will require up to exascale resources.

Support for **GridFTP**[10] would be handy on all clusters because it enables communication between those clusters. One can think of it as an enhanced version of ftp. The most important enhancement concerns the limitations of TCP because TCP is notoriously bad at transferring large data volumes over large distances[11]. When GridFTP is not available we can use SRM (Storage Resource Management). SRM also uses the grid protocol. It can reserve storage space when large volumes of data need to be collected. It also supports data duplication, storage on tape and more.

To provide authentication and authorisation services, support for **voms-client**[12] is necessary at the moment to provide grid access and download using SRM. However, these solutions will be replaced by WebDAV[13] within a year.

The software distribution functionality provided by **CVFMS** (listed previously as a requirement) will be provided by Singularity hub, as all software will be containerised.

PiCaS server (used to distribute tokens for tasks to orchestrate the work in the original architectural plan) will likely be replaced by a more modern work orchestration software, such as Kubernetes.

2.3 Use Case 3: Supporting innovation based on global disaster risk data

The requirements for UC#3 still follow the description given in D4.1.

- The pilot portal is in use (UNISDR)
- As the PROCESS components and their interfaces mature, we will review their suitability to enhance the functionality and usability of the portal
- In parallel to this, we are looking into complementary data sets and data management tools that could help the project reach a broader range of communities that are potential (re-)users of disaster risk data.

A priori we do not expect UC#3 to impose new requirements for the core architecture, as even the largest complementary datasets would not bring resource requirements that exceed the known ones stemming from other use cases. As the third party usage increases, it is possible that PROCESS will receive feedback that allows prioritising the usability improvements to be made to solution components.

2.4 Use Case 4: Ancillary pricing for airline revenue management

The original requirements defined some expectations against the performance in form of Non Functional Requirements, as well. All these requirements remain valid.

The planned framework for the AI part (as presented in D4.1 section 1.4.5 Figure 10) will be realized by an easy to use and easy to set-up machine learning framework. The framework should work both with static data and provide seamless integration with streaming data solutions. For operational reasons and to allow rapid deployment, operating and managing the system cannot require experienced data scientists to perform tasks such as

[10] See GridFTP homepage> http://toolkit.globus.org/toolkit/docs/latest-stable/gridftp/

[11] http://moo.nac.uci.edu/~hjm/HOWTO_move_data.html#_tcp

[12] The Virtual Organization Membership Service (VOMS) is an attribute authority which serves as central repository for VO user authorization information, providing support for sorting users into group hierarchies, keeping track of their roles and other attributes in order to issue trusted attribute certificates and SAML assertions used in the Grid environment for authorization purposes.

[13] The WebDAV protocol provides a framework for users to create, change and move documents on a server. The most important features of the WebDAV protocol include the maintenance of properties about an author or modification date, namespace management, collections, and overwrite protection. See also: https://en.wikipedia.org/wiki/Server_(computing) and https://en.wikipedia.org/wiki/Namespace

enhancing the data training and modelling parts. These considerations differentiate the use case from the UC#1 and explain choosing framework such as H2O.ai and Sparkling Water as starting points. The requirement against the processing platform is to be able to host these frameworks.

After analysing the processing needs for the model training, the HPC-related requirements were refined. As the first approach, our target processing environment will be cloud environment. This approach allows working with a containerization technology with ample operational support. After the early performance testing with the use case prototype, the trade-offs between this approach and a bespoke HPC solution can be reassessed.

Since portability and more flexibility is another important requirement, the use case requires primarily support for Docker containers. Later we might extend this to Singularity containers as well.

The requirement regarding the need for Hadoop ecosystem / HDFS did not change. The data transfer protocol was not yet mentioned in the original requirements, since the source of the airlines ancillary booking data was not yet known at that time. We require the support for SCP transfer in the LOBCDER data acquisition part.

2.5 Use Case 5: Agricultural analysis based on Copernicus data

The aim of the use is to realistically simulate natural processes and impacts of human interventions based on principles, which ensure maximum predictive power. Therefore, it is based on Copernicus datasets fed into the modelling framework PROMET. This software is, as it is described in the deliverables before, closed source. The PROCESS project has gained the ability to execute its binaries within a secured environment on the LRZ resources in Munich.

Unlike announced in D4.1, the pre-processing stage of the Copernicus datasets was also declared to be closed source and therefore it cannot be ported to any other computing resource except LRZ clusters in Munich. For this reason, the workflow presented in Figure 12 in D4.1 on page 42 needs to be adapted. Beside the main program, the closed-source modelling framework of the PROMET environment covers also the Copernicus-Adapter and the PROMET pre-processor. Both will not be directly integrated into the PROCESS ecosystem, but can be configured and described within the PROCESS IEE and will be activated (submitted on the LRZ cluster) via an API directly by any PROCESS end-user. The updated workflow is depicted in Figure 1.

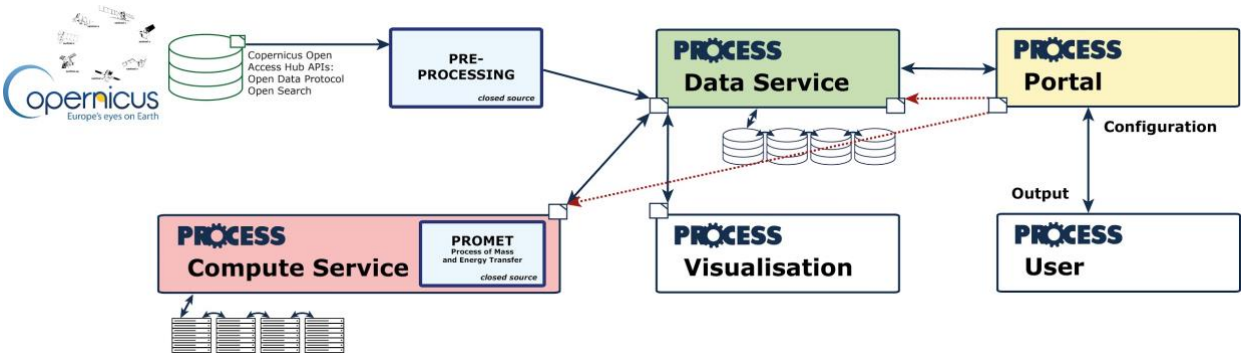

Figure 1: Updated workflow of use case 5.

With these adaptations, the main use case requirements need also to be updated:

- The configuration parameters of the PROMET execution:
  - Time series,
  - Geographical Domain,
  - Simulation Parameters (e.g. physical, meteorological, flora).
- Submit and activate workflow execution via API

- Store PROMET output files for visualisation and download

The described software requirements in D4.1 will be reduced to a submission and output transfer API. Hardware, security and privacy requirements remain unchanged.

2.6 Common requirement analysis[14]

Hardware Requirements

The common hardware requirements remain unchanged as presented in section 2.2 in D4.1.

Software Requirements

Most of the common software requirements remain unchanged. The updated requirements of the use cases suggest two changes to the previous common requirements. First, the data transfer protocol requirements have become more specific and require SCP and GridFTP access. Second, UC#5 shows a requirement that was not present previously and can be generalized for similar future use cases. Since UC#5 relies on closed source, a proxy API to communicate with the use case software needs to be developed. This API can be integrated with the IEE to support future use cases with similar requirements regarding closed-source software.

In addition to section 2.2 in D4.1 the new software requirements are:

- Support different protocols like SCP and GridFTP for data transfers
- Generalized proxy API for closed-source use cases

2.7 Adaption of Requirements

Based on the updated use case requirements analyses in the sections 2.1 to 2.5, we evaluate, if any adaptations to our services and conceptual models are necessary. If this is the case all changes will be presented in this section.

2.7.1 Common conceptual model[15]

In D4.1, we presented the first approach towards the final common conceptual model. With the updated requirements of the use cases, no adaption regarding this model is necessary. It will be again evaluated within D4.4.

2.7.2 Extremely Large Data Service-oriented Infrastructure

The Extremely Large Data Service-oriented infrastructure allows connecting user's dispersed data sources while handling data staging/movement between HPC sites. It exposes staging as a service to the IEE where data staging requests are queued and performed asynchronously. IEE is notified through webhooks on the completion of data staging. This allows, IEE to coordinate execution with data staging. The architecture allows for different data staging backends, which are meant to be application specific.

Below, we summarize the fulfilment of the requirements for this platform coming from changes to the use case requirements:

Use Case 1

- SCP is currently included as a supported protocol. The data staging service is capable of doing direct copies between sites using SCP.

[14] The common requirement analysis from the deliverable D4.1 is provided in Appendix A.

[15] The common conceptual model from the deliverable D4.1 is provided in Appendix B.

Use Case 2

- GridFTP is not yet integrated into the staging service. While protocols such as GridFTP are important due to their performance, other protocols with equal performance but less complexity such as FTS3 are being investigated as a viable alternative.
- SRM is currently supported but not fully integrated in the staging service. This mechanism uses a data job approach whereby a special job is submitted to the HPC site to pull in data over SRM.

Use Case 3

- No update

Use Case 4

- SCP is currently included as a supported protocol. The data staging service is capable of doing direct copies between sites using SCP.
- HDFS is not currently included on the data side. The requirement of HDFS is more applicable to the computing side since it forms part of Hadoop computing infrastructure.

Use Case 5

- No update

2.7.3 Extremely Large Computing Service-oriented Infrastructure

The Extreme Large Computing Service-oriented infrastructure allows smooth scheduling of computations on large HPC infrastructures composed of multiple HPC Clusters (Sites). Its main component Rimrock is used for interaction between the upper layer – IEE (see the next chapter) and the low layer composed of HPC resources.

Below, we summarize new requirements for this platform coming from changes to the use case requirements:

Use Case 1

- **Transition from Docker to Singularity Container** - Docker container would not be feasible for the HPC, therefore this change allowed Rimrock to be used for UC#1 codes scheduling. As Singularity is executed by typical batch script, Rimrock may be used as-is without the need for further adaptation. Specific aspects of Singularity are handled on the upper layer by IEE.
- **Data transfer has to follow the SCP protocol** - computing platform is not directly involved in data transfer as it is handled by IEE and data infrastructure, thus there are no requirements for the computing platform for this aspect.
- **Support for specific versions of tools and libraries** – it will be provided due to the use of containerization technology (Singularity).

Use Case 2

- **New UI requirements** - are a non-factor for the computing platform and will be handled by IEE updates mentioned later in this document
- **Need for Singularity container** - requirements are the same as for UC#1
- **Requirement for fat nodes** - Rimrock may pass any attributes to the queuing system

such as SLURM via batch script, therefore we may adapt to the need of the application by selecting appropriate partition (e.g. big mem) as needed

- **Data transfer** is out of the scope of the computing platform.
- **Horizontal scaling** - Rimrock should scale sufficiently for this use case – if we detect higher scalability needs during the project, we can deploy multiple Rimrock instances, to cover such scenario.

Use Case 3

Is not providing new requirements for computing platform at the moment.

Use Case 4

- **Switch from HPC to Cloud and Docker for the first prototype** - this change means that the HPC computing part including Rimrock will not be used for this use. Code will be run from IEE via Cloudify.

Use Case 5

- **Closed-source components prompting API driven computations** - as the access to the code is restricted, Rimrock cannot be used to run it for this UC. In turn, IEE will directly interact through an API with a dedicated job-scheduling component on the LRZ cluster provided by the LMU team. The component will be considered as a Computing Infrastructure component in place of Rimrock, but just for this specific Use Case.

2.7.4 Service Orchestration

Service orchestration is often understood as a process for automated configuration, deployment and other management activities of services and applications in the cloud. It can automate the execution of different service workflows including deployment, initialization, start/stop, scaling, healing of services based on standardized descriptions of composed services, relations between components and their requirements. In the PROCESS project, we use the OASIS TOSCA standard for service description and Cloudify for orchestration. In the next section, we describe how the service orchestration can fulfil the new requirements from use cases.

Use Case 1

The updated requirements are focused on HPC environments. However, the service orchestration framework can deploy similar execution environments with requested software/hardware components (Docker, GPU, SCP connection, deep learning libraries) in Cloud for running the use case.

Use Case 2

The updated requirements are focused on execution in HPC environments. The service orchestration can deploy transfer nodes on demands.

Use Case 3

No changes in requirements.

Use Case 4

Execution environments can be deployed in Cloud via service orchestration. However, due to the overhead of virtualization layers, HDFS may not reach the performance as on bare metal.

This limitation is set by the underlying technologies, what is out of the scope of service orchestration.

Use Case 5

Service orchestration can deploy base hardware/software environment in the cloud and users can add closed-source software on top of the deployed environment. Therefore, the requirements of using closed-source software are fulfilled.

Security and privacy policies are defined by cloud providers and are out of the scope of the service orchestration framework. Users can first choose the cloud provider that can fulfil the security and privacy requirements of the use case, then use the service orchestration framework to deploy execution environment in the provider's cloud.

2.7.5 User Interface

The primary user interface providing access to PROCESS resources is the IEE - Interactive Execution Environment, which is being implemented in the context of WP6. IEE is meant as the topmost layer enabling PROCESS use case operators to interact with the infrastructure without coupling with low-level communication protocols and direct deployment of computational jobs to HPC and cloud infrastructures. Below sits a layer of services capable of interacting with hardware resources. These services expose an API, which in turn is accessed by the IEE. The API may be accessed by other custom tools deployed on behalf of PROCESS infrastructure users. The abovementioned architecture was briefly summarized in the D6.1 deliverable (in Figure 1).

Given the strong diversity of PROCESS use cases and the corresponding requirements, the primary design goal of IEE was to ensure that it remains generic and may be used to invoke various types of computational tasks on extensive distributed computing infrastructures. As already described, IEE is conceptually based on Model Execution Environment developed within the EurValve project, which fulfils a similar role (albeit on a much smaller scale), and is being extended with exascale deployment capabilities in the context of PROCESS, while also being improved in terms of its TRL.

Regarding specific use cases and their associated requirements, IEE addresses specific points formulated in the Updated requirements analysis section (Sections 2.1 – 2.5) of this document in the following manner:

Use Case 1

- **Support software environment containers with particular regard to Singularity containers, which are suited for HPC clusters.**

  This requirement has been addressed by container management layer in the context of IEE - an extension to its underlying services layer, enabling management, deployment, and polling of container-based computations on arbitrary HPC resources. As a result, containers supplied by use case developers (not just in the context of UC#1) may be wrapped into suitable Pipeline Steps and arranged into Pipelines for execution on computing clusters.
- **Data transfer has to follow the SCP protocol.**

  In parallel with enabling deployment of containers to HPC sites, integration with the PROCESS data services is affected by implementation of so-called Stage-In Steps and Stage-Out Steps, permitting data to be marshalled prior to commencement of computations. Since downloading the required data may take a long time, the stage-in/stage-out process is fully monitorable and can be visualized in the IEE.

Use Case 2

- **User-friendly UI for selecting the data and workflows. Mechanisms for launching workflows are being included in this component. The latter can also be used to monitor the jobs.**

  IEE addresses all the above needs by provisioning a way to launch LOFAR computations while specifying the required input parameters (along with the target datasets) and automating all the data stage-in operations while enabling users to monitor on-going computational jobs. Further UI customization for UC#2 is foreseen at later stages of the project.
- **Possibility to use Singularity containers as workflow steps is still desired**

  Similarly to UC#1, Singularity is fully supported by IEE and prototype LOFAR Singularity containers are included in the initial release of the environment.

Use Case 3

While no specific requirements have been formulated by UC#3 with regard to the features of the end-user interface, generic nature of IEE ensures that any computational services associated with this use case can be integrated with the infrastructure as extensions to the IEE services layer.

Use Case 4

- **After analysing the processing needs for the model training, the need of HPC was revisited again: as a first approach our target-processing environment will be cloud.**

  While the initial prototype of IEE focuses on deploying application containers to HPC resources (as batch jobs), it has been agreed that IEE will be integrated with the Cloudify environment, enabling it to treat cloud sites as additional resources to which PROCESS containers can be deployed. This integration is foreseen to take place during the second phase of the project (in the context of managing multi-site computations from a single entry point).

Use Case 5

- **Beside the main program, closed-source modelling framework of the PROMET environment covers also the Copernicus-Adapter and the PROMET pre-processor. Rather than directly integrate them into the PROCESS ecosystem, they will be configured and described within the PROCESS IEE and activated (submitted on the LRZ cluster) via an API directly from any end-user of PROCESS.**

  UC#5 has specific deployment requirements regarding the security of the application code. Therefore, it is expected that no container will be released by this use case for deployment to arbitrary computing resources. The quoted requirement encapsulates the result of discussions with representatives of UC#5 regarding management of PROMET computations via the IEE. It has been agreed that a dedicated service will be implemented for this purpose, and IEE will communicate with this service to schedule computations while externally treating them as Pipeline Steps, similar to those present in other use cases. Implementation of the above features is on-going.

2.8 Related deliverables

- D4.1 - Initial state of the art and requirements analysis, PROCESS architecture
- D2.1 - Progress Report (UC#1-5)

3 Validation of the first prototype

Validation is carried out in two steps: first, focusing on functional requirements and second, on non-functional requirements, essentially, performance evaluation. In the first phase, we check that use case applications meet their requirements as defined in D4.1 and in updated sections 2.1 through 2.5 of this deliverable. We also check whether PROCESS platform reaches TRL5 by validating its components and their interactions. In the second phase, we use a purposely-built validation container to study PROCESS middleware scalability. Specifically, we use this container to estimate the overhead associated with using the PROCESS platform by taking measurements in different scenarios as defined in D3.1. Based on those measurements, we build a model to extrapolate PROCESS middleware performance behaviour as the data and/or computation tend towards the exascale.

3.1 Validation process

Validation has been performed on the Prometheus cluster in Cyfronet, Krakow. This cluster was built on purpose by Hewlett Packard Enterprise for HPC tasks and is based on HP Apollo 8000 (2160 nodes of ProLiant XL730f Gen9 and 72 GPU-enabled nodes of ProLiant XL750f Gen9 supplemented by 3 fat nodes (768 - 1536 GB of RAM/node) of ProLiant DL360 Gen10). The platform has the 53604 Intel Xeon (Haswell/Skylake) CPU cores for generic calculations, 144 GPGPUs (NVIDIA Tesla K40 XL), 282 TB of RAM and c.a. 10 PB of storage space. The cluster utilizes high-performance Infiniband interconnect (FDR 56Gb/s). The total theoretical computation power is rated at 2403 TFlops.

The Prometheus cluster is running Linux Operating System (CentOS 7). For the job scheduling SLURM system is used. The system allows optimal utilisation of the resources by defining separate partitions for a different type of jobs (test, short, normal, long) with dedicated special-purpose partitions (such as GPGPU or BigMem for fat nodes). Additionally, the storage mentioned above is organized into multiple pools such as user homes (NFS based rated at 1 GB/s), permanent group directories (on Lustre rated at 30 GB/s) and fast but temporary (up to 30 days) SCRATCH pool (also Lustre - rated at 120 GB/s).

3.1.1 Validation of functional requirements

Below, we consider all the requirements as initially defined in deliverable D4.1 and updated in section 2 above. For each use case and requirement, we check whether it is fulfilled or it has on-going development.

Use Case 1

- **Support for Singularity containers instead of Docker as initially stated**

  Singularity containers can run as batch jobs on HPC through their workload management systems supported by Rimrock (see section 2.7.3). Furthermore, a container management layer has been developed in the context of IEE (see section 2.7.5).
- **Environment for managing distributed training jobs life cycle**

  The container management layer in IEE enables management, deployment, and polling of container-based computations on arbitrary HPC resources (see section 2.7.5).
- **Distribution of datasets across multiple computing sites with fast connectivity and low latency**

  At this stage of the project, only single site (Cyfronet) is targeted; this will be addressed in the next updates.
- **Parallel distributed dense linear algebra and multi-GPU settings**

  Parallel LA integration is ongoing and should be provided by appropriate libraries; for multi-GPU setup, see Uber Horovod below.

- **User-friendly environment for managing UC life cycle, from selecting input to getting output**
  This requirement is fulfilled through IEE (see section 2.7.5).
- **Software requirements: TensorFlow, Keras, Theano, PyTorch, Openslide, ASAP, DICOM, Spark, Hadoop**
  These requirements are fulfilled through containerization.
- **Hardware requirements: GPUs, large memory, fast interconnect, caching**
  These requirements are fulfilled by the use of Prometheus (see description above).
- **Security and privacy: need to anonymize data before use**
  This requirement is fulfilled by the use of publicly available data without the need of anonymization for development purposes. Future developments will involve the use of anonymised data to expand dataset sizes.
- **Flexibility in order to exploit regular software updates**
  Requirement is fulfilled through container utilization (see section 2.7.5); furthermore, IEE provides tagging system, which allows using any version of any software in each pipeline step.
- **Support for Uber's Horovod**
  Uber's Horovod is under integration at the local infrastructure level.

Use Case 2

- **User-friendly environment for UC life cycle management**
  As for UC#1, this requirement is fulfilled through IEE (see section 2.7.5).
- **Support for Singularity containers as workflow steps**
  Singularity containers can run as batch jobs on HPC through their workload management systems which is supported by Rimrock (see section 2.7.3).
- **Use of special computing ("fat") nodes for some processing steps**
  This requirement can be fulfilled through Prometheus and SLURM (see section 2.7.3) for later updates.
- **Mechanisms for efficient transfer of extremely large datasets between sites**
  Use of GridFTP (see below) or use of data transfer nodes (see sections 2.2 and 2.7.2).
- **Significant horizontal scaling of computing resources**
  This requirement can be partially fulfilled through Prometheus by using multiple instances of Rimrock (see section 2.7.3) and/or using multiple sites.
- **Software requirements: GridFTP, VOMS, SRM, WebDAV**
  GridFTP is not fully integrated yet, simpler alternatives, yet equally capable such as FTS3 are considered (see section 2.7.2); SRM supported but not integrated; WebDAV not addressed or not documented.
- **Other hardware requirements: fast interconnect (1-10Gps), sufficient storage (20TB) and GPUs later**
  Provided through Prometheus (see section 2.7.3) and other computing resources providers for later updates.

Use Case 3

- **Easy-to-setup and use and extensible UI with data discovery functionality**
  Provided by IEE (see section 2.7.5).
- **Support for multiple indexing/curation approaches**
  Provided through containerization.

Use Case 4

- **Support for Hadoop/HDFS, HBase, Spark, TensorFlow**
  Provided mostly through containerization; HDFS is not containerized, but it is accessible through adaptor which is part of a micro-infrastructure.
- **Data store for < 100TB**
  Provided by Prometheus.
- **Compliance with EU GDPR**
  In the first stage we rely only on generated data, no real personal data is used.
- **Use of cloud infrastructure**
  Provided by Cloudify integrated into IEE.
- **Support of streaming data in addition to static data; use of H2O and Sparkling Water and Docker containers**
  Provided through containerization.
- **SCP support for data transfer**
  SCP is supported by Data Services (see section 2.7.2).

Use Case 5

- **API for submitting jobs to PROMET on LRZ and for retrieving results**
  Under development.

Given the lack of agreed upon assessment methods for TRL levels, especially for software development, we stick to EU's definition of TRL 5. Consequently, to show that PROCESS middleware as a whole reaches that level, we show that its enabling technologies (i.e., the underlying services), reach or exceed the same level and that they are integrated and work well together in the targeted operational environment.

In the state of the art of technologies used in PROCESS in deliverable D4.1, we showed that those technologies either match or exceed TRL6. Although, some of them, notably LOBCDER and IEE, have undergone some heavy lifting, they still match or exceed TRL6 as they are demonstrated to run on Prometheus, which is part of the expected operational environment. Furthermore, as evidenced in previous subsection, most UCs, especially, UC#1 and UC#2, which are target for the first PROCESS platform prototype, meet their requirements and run on the platform. With the different services being integrated, we can confidently assert that PROCESS platform reaches TRL5.

However, there are a few issues to be addressed. Although, UC#1 runs on PROCESS platform, the multi-GPU setup, which is important to the use case performance, is not yet integrated. Similarly to UC#1, UC#2 runs on the platform but requires extreme large datasets be transferred from temporary locations to computing sites in an efficient way using either GridFTP or equally capable alternatives and this is not yet in place. Furthermore, albeit they are not targets for the current platform prototype, UC#4 and UC#5 also raise some issues: they pertain to the lack of direct support for HDFS and non-checking of GDPR compliance, as of now, for the first use case and the yet-to-start development of the job submission API for the second use case.

3.1.2 Performance evaluation of PROCESS platform

After evaluating the functional requirements of the process platform, we now evaluate an important non-functional requirement, which is PROCESS middleware scalability. For this evaluation, a dedicated pipeline composed of stage-in and sage-out steps with in between a computation step materialised by a dummy validation container mimicking basic operations of a standard computational task. Specifically, this validation container checks whether input and output directories are provided and then sleeps for a specified amount of time. The "raison d'être" of this pipeline is to make it possible to estimate the overhead associated with PROCESS services and their interplay as defined in D3.1. An essential point arising from the

latter is that, in order to show the scalability of PROCESS approach, this overhead should stay either constant or increase only marginally as we increase the number of running containers, thus the volume of data and intensity of computation. The simplicity of this container helps in keeping the focus on PROCESS services and their interaction instead of the actual use cases whose individual behaviours are known. The validation pipeline is illustrated in the image below.

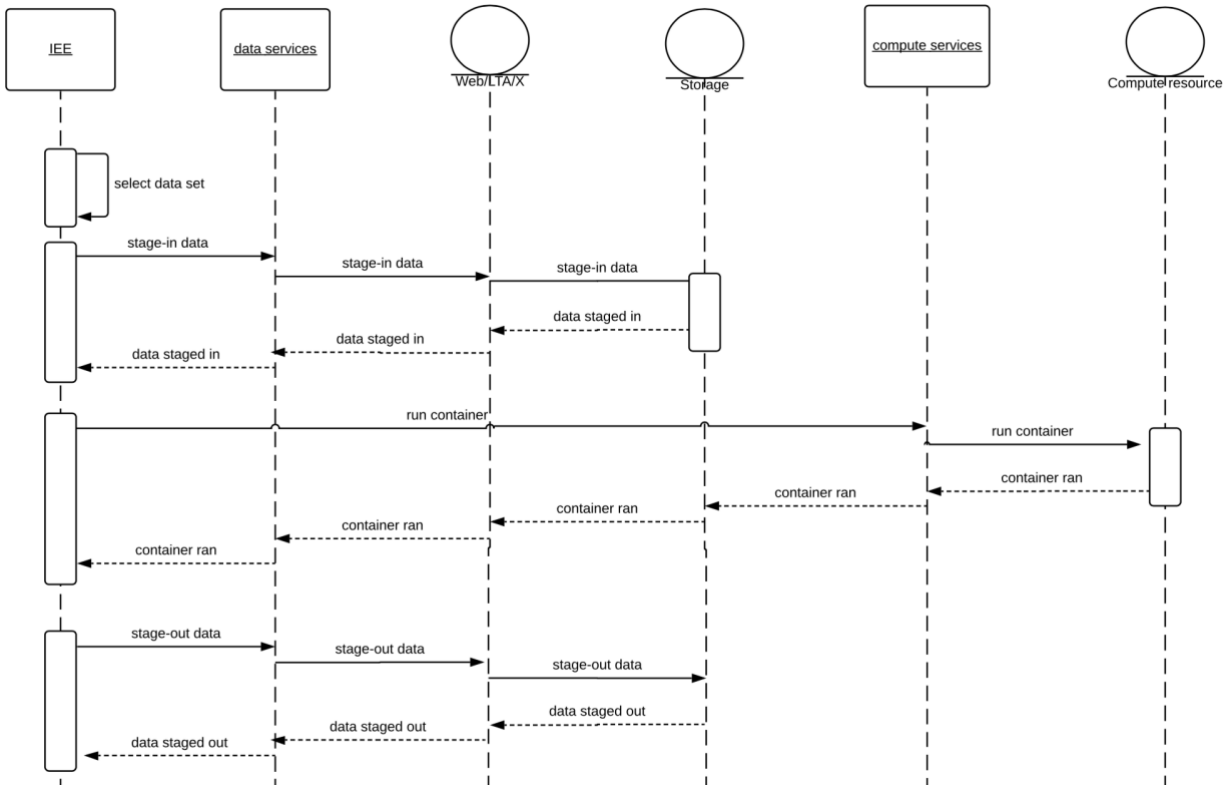

Figure 2: Validation pipeline.

We decide to use a small data set to be able to perform several test runs in a short time. This test data set of about 1 GB represents measurement sets from UC#2. Of course, more representative data sets will be used in later updates. Among the 8 measurands defined in D3.1, we measure those contributing to overhead of PROCESS platform, namely T1, T2, T4 and T7, and also T3 and T8 pertaining to staging. T1 and T2 pertain to IEE, T4 to Rimrock, SLURM and finally, T7 to LOBCDER. Consequently, each of these components are instrumented to take the appropriate timing measurements. It appeared that T1 and T2 correspond to the same aggregated action within IEE, leading to just one measurement making up T1. We also drop T4 as it currently measures queueing time in SLURM which is not part of PROCESS but rather of Prometheus HPC environment.

At this stage, we can only perform the first two scenarios defined in D3.1 for performance evaluation. In the first case, we deploy a single container on Prometheus. Then, in the second, we increase the number of containers, but still on the same cluster. The number of containers to deploy is chosen to have a uniform coverage of the space of sizes allowed by the available data and reasonable actual evaluation time. For instance, for UC#2, we know we can go up to 1800 containers if each container reduces one observation; instead, we sparsely sample this space of 1800 values down to a handful runs. Due to the architecture of the Prometheus cluster, where each node consists of 24 cores, we allocate containers in multiple or divisors of 24 or 10. Consequently, we run setups with 1, 2, 10, 16, 40, 48 and 240 parallel containers. The results are shown in the figures below.

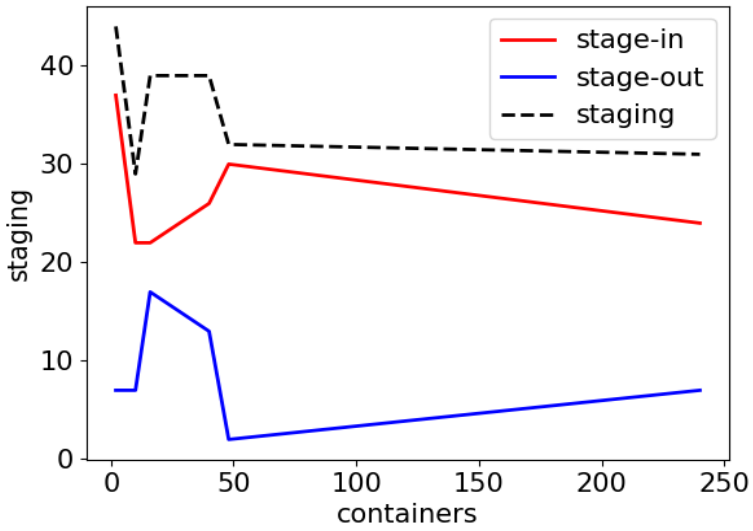

Figure 3: Scalability of PROCESS platform staging.

From the Figure 3, we observe that for a given fixed data set, staging time is not correlated to the number of containers being submitted to the HPC system. Indeed, after an erratic behaviour for smaller values up to 50, as the number of containers is increased, the aggregated staging time (dashed line) stays constant. Consequently, staging does not harm scalability. The difference in values from stage-in and stage-out is due to the fact the former includes wide-area network transfer times from wherever the data are located to Prometheus while the latter concerns only intra-site transfer times. Next, we study the overhead associated with using PROCESS currently captured in T1 and T7 and shown in the figure below.

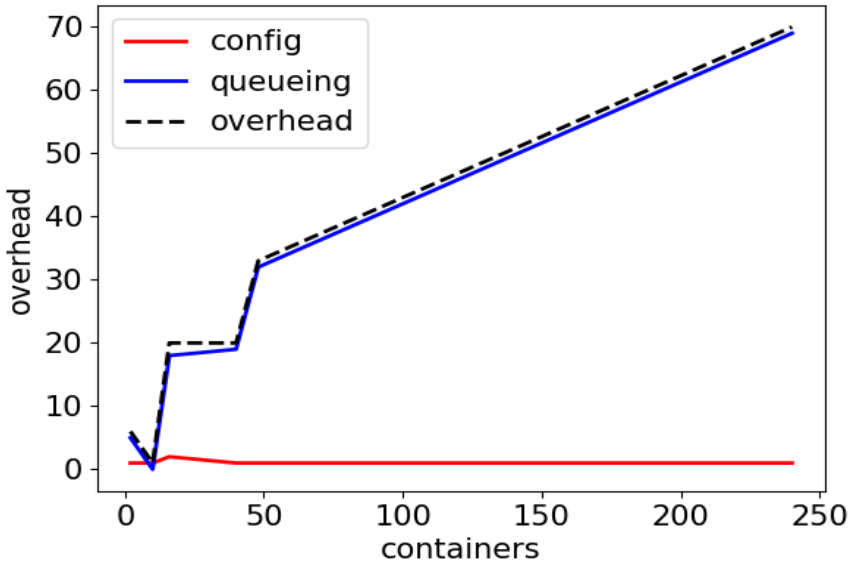

Figure 4: Scalability of PROCESS platform overhead.

Here also, we observe that after a steep increase from 10 to 50 containers, the increase becomes moderate after 50 onward. To do the extrapolation for analysing scalability of the PROCESS ecosystem, a statistical/machine learning approach is used. Because of the prohibitive execution or data transfer times of some of our UCs, the performance evaluation of PROCESS is considered a high-dimensional problem where the number of runs (samples) is in the same order as that of predictors (measurands). As approaches using

random forest are well-known to deal well with these cases (e.g., see Genuer's paper[16]), we first tried one of those. Unfortunately, the collected data set is too small to derive any insight from the data as the generated model is too poor. Turning to simple correlation analysis, we observe a significant positive correlation between the overhead and the number of containers as shown in the figure below.

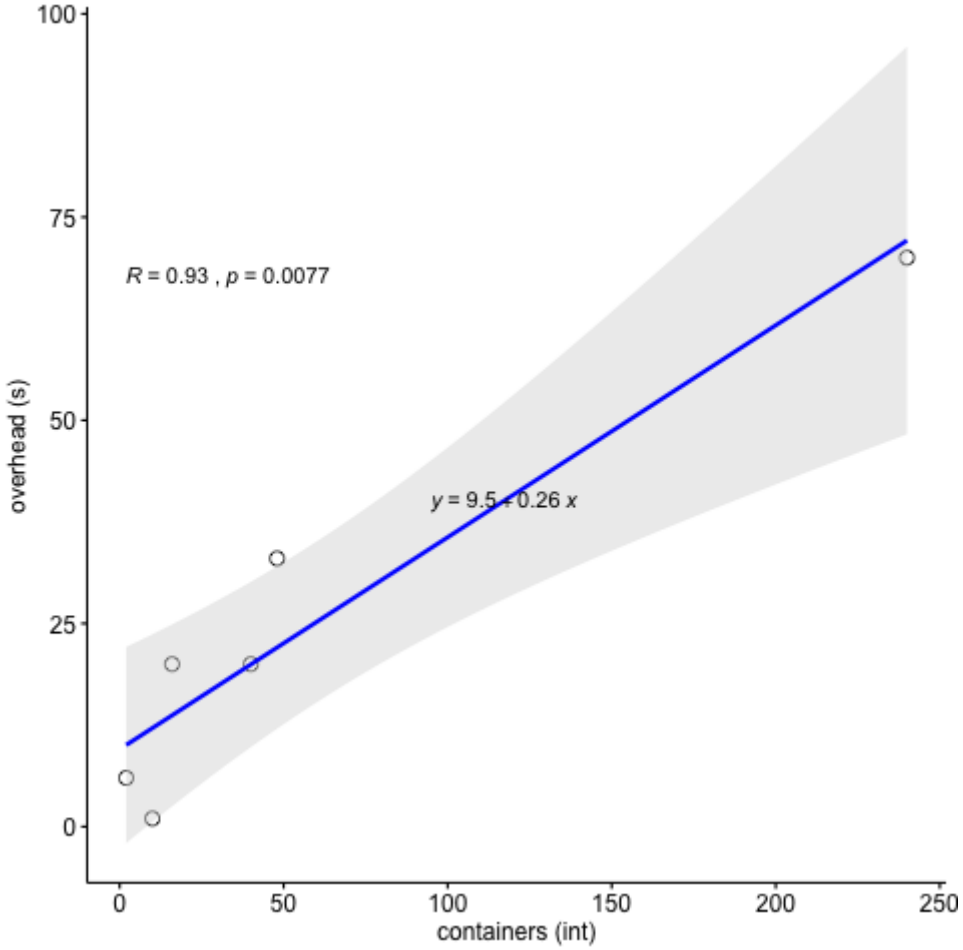

Figure 5: PROCESS platform overhead model.

The graph is a scatter plot representing the PROCESS platform overhead as a function of the number of containers used. Shown R and p values are respectively the correlation coefficient and the p-value of the correlation test. Values for R are between -1 and 1, with values close to 1 (our case) meaning positive correlation (both variables increase together); the p-value captures the significance of the correlation, which is significant whenever $p<0.05$ which is the case here. The light grey area represents the 95% confidence interval around the regression line (blue) modelling the covariation of both metrics. Although the model shows a linear relationship (visible in the equation on the regression line), a small slope value means the overhead only moderately increases as the containers increases. For instance, if we assume every container performs reduction for an entire observation for UC2, we would require 1800 containers to reduce the entire LTA, leading to an overhead of about 477 seconds.

3.2 Validation conclusion

We evaluated PROCESS platform prototype using a two-step approach: we first checked whether functional requirements are met and second, we assessed the platform performance. As shown in section 3.1.1, most of the PROCESS components/services run in the pilot environment, are integrated and meet most of the UCs requirements. Especially,

[16] https://hal.inria.fr/inria-00340725/document

UC#1 and UC#2, which are targets for the prototype. We show that the platform as a whole reaches TRL5. In section 3.1.2, we analyse performance of the platform prototype, focusing on the overhead induced by using the platform. Our intent is to show that this overhead is not detrimental to PROCESS scalability and it appears that this is, indeed, the case. We also observed a couple of issues, mostly pertaining to non-functional requirements, which will be addressed in the follow-up updates.

Furthermore, PROCESS architecture matches the definition of a distributed services platform (DSP) as recommended by the Big data and extreme computing (BDEC) group for platforms targeting the exascale[17]. In addition, although PROCESS does not include any edge/fog computing services, our choices of using Kubernetes for the data services and containerization fit with the conclusions and recommendations of the BDEC report.

3.3 Related deliverables

- D3.1 - Performance modelling and prediction
- D4.1 - Initial state of the art and requirements analysis, PROCESS architecture
- D4.2 - Report on architecture evaluation and Dissemination
- D6.1 - First prototype
- D5.1 - Design of a data infrastructure for extreme-large datasets
- D5.2 - Alpha release of the Data service

4 Reference exascale architecture

Reference exascale architecture addresses the requirements coming from the requirement analysis (which was initiated within deliverable D4.1 and updated in this deliverable) as well as the validation of the first prototype (see section 3). Its design was initiated during the preparation of the deliverable D4.1 where it is called *"functional design of the PROCESS architecture"* (see Figure 19 in Section 3.1 of the deliverable D4.1).

The aim of the proposed reference architecture is to characterize key attributes and properties that have to be handled by every scientific application using exascale data and computations. From altogether viewpoint, the reference exascale architecture (see Figure 6) is divided into the following parts (from top to bottom):

- **Users of the exascale scientific applications** (in yellow) - the exascale system has to support functionalities required by its user communities. That also means to support legacy applications in some cases (see the Copernicus use case). According to the initial and updated requirements analysis, the best way is to provide a containerized application repository. It is flexible, scalable, reusable and ready to use. Moreover, it does not require any special technical skills (especially, related to integration - exascale data processing is often contingent on complex software tools involving expert knowledge about its management) to make it run on the resource infrastructure (see the LOFAR use case).
- **Virtualization layer** (in blue) - interoperability of data infrastructure and computing is the key and critical requirement of the exascale systems. To use both infrastructures in the most efficient way, we propose the exascale reference architecture based on containerisation instead of virtual machines. The performance of the infrastructure as the whole is utilized in a better way. It is caused by minimization of overheads (e.g. software duplications). Thus virtualization layer based on containerization approach

[17] M Asch, T Moore, R Badia, M Beck, P Beckman, T Bidot, F Bodin, F Cappello, A Choudhary, B de Supinski, E Deelman, J Dongarra, A Dubey, G Fox, H Fu, S Girona, W Gropp, M Heroux, Y Ishikawa, K Keahey, D Keyes, W Kramer, J-F Lavignon, Y Lu, S Matsuoka, B Mohr, D Reed, S Requena, J Saltz, T Schulthess, R Stevens, M. Swany, A Szalay, W Tang, G Varoquaux, J-P Vilotte, R Wisniewski, Z Xu and I Zacharov> "Big data and extreme-scale computing: Pathways to Convergence-Toward a shaping strategy for a future software and data ecosystem for scientific inquiry" The International Journal of High Performance Computing Applications 2018, Vol. 32(4) 435–479

exploits the infrastructure resources in the most optimal way and also it supports the requirements from our user communities.

- **Virtual File System** (in green) - requirements coming from the exascale scientific applications could be divided into two main groups: distributed data federation, and metadata. It is very common that the exascale scientific applications have highly complicated datasets, that need to handled and processed by relevant systems. For that purpose, its file systems must have a module capable to work with metadata. The metadata module has to be federated and distributed as well as the management system for the data infrastructure itself. At this level of the infrastructure, the system architect has to be careful whether the component will be containerized, or not. On the one hand, the exascale system has to avoid overhead and latency (according to our experiments, it is caused by needless duplication of software) thus we prefer containers to virtual machines. However, on the other hand, all infrastructural services do not need to be virtualized. For example, virtualization of HBase (through containers, or virtual machines) is not necessary because it leads to dataset duplication in the worst case or a performance overhead in the best case. Thus a better approach is to support infrastructural services ecosystem through micro-services. Micro-services serve as adapters and connectors to infrastructural services. They are integrated into a containerized micro-infrastructure, which is customized according to requirements coming from a use case and connecting them to a distributed virtual file system. The micro-infrastructure allows for application-defined infrastructures with the main advantages being threefold: First, services can be customized for the application; e.g., data staging service. Second, minimizing global state management (a major scaling issue); e.g., instead of having one global index for all files for all applications, have micro-infrastructures manage their own local indices and states. Third, micro-infrastructures are isolated from each other, which increases security between users of different applications. The PROCESS distributed file system layer needs to be virtualized because it has to run on top of multiple file systems. Also, it is crucial that access to a data storage federation is unified. Thus the virtual file system is distributed.
- **Interactive Computing Environment** (in red) - this part of the infrastructure is related to scheduling and monitoring computing resources. The infrastructure has to be loaded as balanced as possible. Two kinds of resources have been recognized as suitable for exascale scientific applications, namely: high performance computing (HPC) resources, and cloud resources. HPC manager is based on a queuing approach. Manager of cloud resources is based on REST API. Both types of resources are often enriched by support from high-throughput resources or accelerated resources. For example, GPU utilization within machine learning and deep learning application is very common practice.

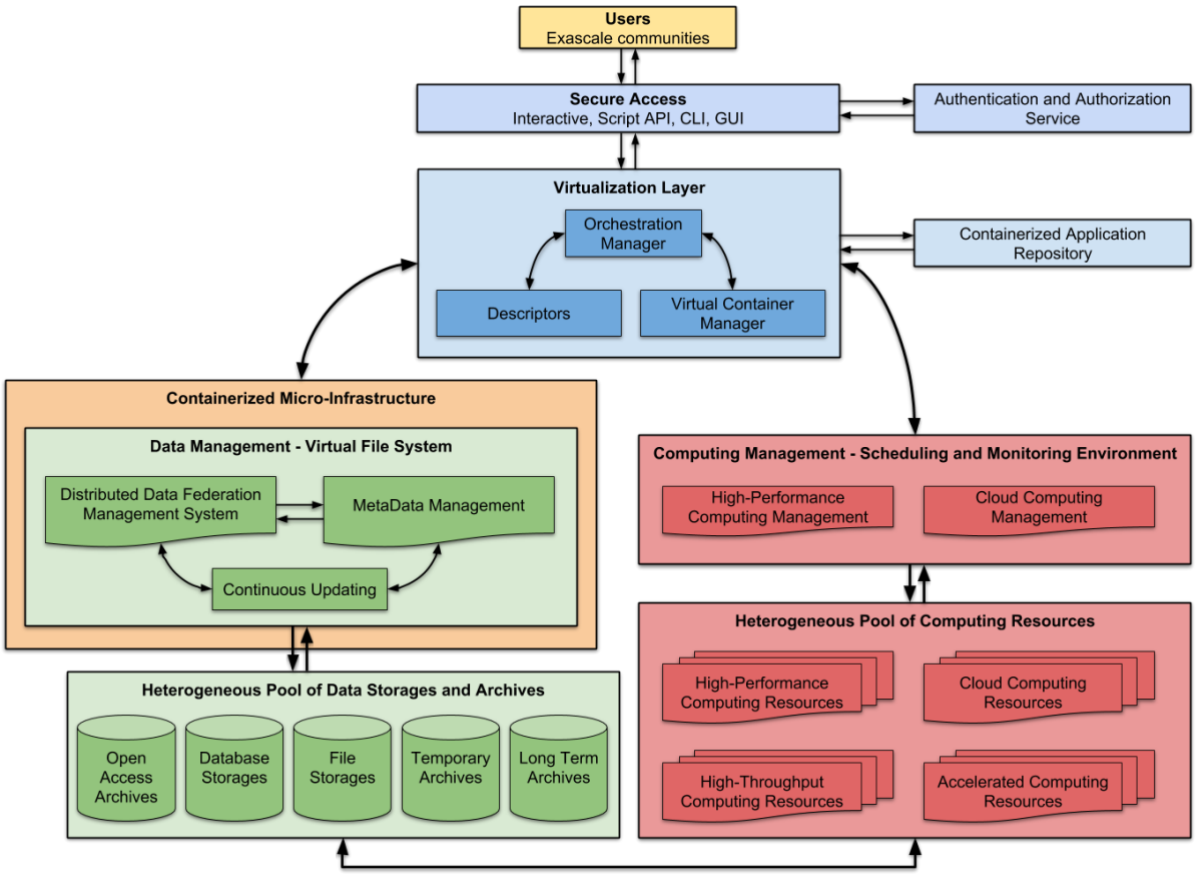

Figure 6: Reference exascale architecture.

4.1 Related deliverables

- D4.1 - Initial state of the art and requirements analysis, PROCESS architecture
- D5.1 - Design of a data infrastructure for extreme-large datasets
- D4.2 - Report on architecture evaluation and Dissemination
- D5.2 - Alpha release of the Data service

5 Updated PROCESS architecture

In Deliverable 5.2, we described two scenarios derived from two out of the five PROCESS application use cases, namely UC#1 and UC#2 with the aim to use them as a first step to validate the PROCESS architecture proposed and described in Deliverable D5.1. Since we are just a half way of the project, and we have just released the Alpha version of the PROCESS data service in Month 15, we are not aiming at validating a full scale use case but rather a proof of concept (prototypes) that can show that the proposed architecture is functional and has all the features that could help us to progress and reach the final goals in the second phase of the project. In validation of the PROCESS architecture, we aim to validate the following points:

- Integrated software components (LOBCDER, DataNet, Rimrock, and Dispel) developed in various WPs are working together as expected
- The portability of the PROCESS architecture, whether the proposed architecture, developed and tested only at local datacentre scale, can be easily deployed across

the geographically distributed datacentre; we will start with two datacentres (SURFsara in Amsterdam, and CYF in Krakow)

- The usability of the proposed architecture through the scenarios derived from real use cases; we want to engage with end users and check how they will use and be interested in the current implementation of the PROCESS infrastructure

We believe that this three-point checklist will give us enough insight in the proposed architected in order to continue with the work in the second phase of the project. The **integration** checks will help our development team to work together and develop a team work attitude, the suitability test will help us to keep our users in the loop and further tune the application requirement, and finally, the **portability tests** will help use to easy is harvest computing and storage resources needed for real exascale applications.

In the rest of this section, we describe the steps we have followed to implement the two use case scenarios.

5.1 Scenario 1: UC#1 development

The core component of this workflow is a composition of **two main tasks** which have been containerized to facilitate dynamic movement of compute between compute infrastructures. The following text describes the containerization effort of the separate tasks:

- **Task 1**: The functionality of this task can be considered as a pre-processing task, which pre-processes public datasets CAMELYON16 and CAMELYON17 to output a 32GB hdf5 file. The source code is in Python with specific library dependencies, parameters, and versioning. Containerization can capture all these dependencies and allow for better portability of the code. In our current examples, we consider two containerization technologies: Docker and Singularity. Docker image will allow the task to execute in cloud-native environments such as Kubernetes while a Singularity image will allow the task to run on traditional HPC clusters. For this reason, every task is compiled into two images.
  - **Docker image**[18]: this image is based on ubuntu:16.04, which is configured with apt-get and pip to install the required dependencies. Execution code is added to the container at compile time while input data is mounted at execution time.
  - **Singularity image**[19]: this image based on ubuntu:16.04, which is configured with apt-get and pip to install the required dependencies. The nature of Singularity also requires environment variables to be set. Also, execution code and data are handled differently in Singularity. Singularity containers have direct access to user's home directory. For this reason, the code is not added at the compile time but instead, it is loaded during the container execution. This is achieved by exposing Python interpreter as the container's default execution environment whereby the Python code can be passed as a parameter at the execution time. Similarly, Singularity container assumes that data is also present on a reachable host path without mounting. The path of the input data is passed as a parameter during execution.
- **Task 2**: The functionality of this task is considered as the core of the workflow and

[18] The Dockerfile for this task is found at:
https://github.com/recap/MicroInfrastructure/tree/master/ConatinaerAdaptors/uc1/stage-1

[19] The Singularity for this task is found at:
https://github.com/recap/MicroInfrastructure/tree/master/ConatinaerAdaptors/uc1/stage-1

takes input from the task 1 and other inputs to train a neural network accelerated by GPUs. This hardware dependency means that the containerization is not straightforward. The source code is Python based for specific hardware, library dependencies, parameters, and versioning. Containerization can capture all of these dependencies with some workarounds and also include hardware dependencies of containers. In our current examples, we consider two containerization technologies: Docker and Singularity. Docker image allows the task to execute in cloud-native environments such as Kubernetes while a Singularity image allows the task to run on traditional HPC clusters. For this reason, every task is compiled in two images. The hardware aspect of this task makes the Docker and Singularity approach a bit different.

- **Docker image**[20]: Access to GPU hardware through Docker relies on runtime plugins, which expose the hardware to virtualized containers; i.e., nvidia-docker. The Python code depends on specific Nvidia CUDA and cuDNN versions and libraries. For this reason, the container is based on Nvidia images with these specific dependencies already incorporated. Other libraries are installed at the compile time using apt-get and pip. Also here, the code relies on specific versions of the Tensorflow and Keras libraries. The code is compiled into the Docker image while input data is mounted at the runtime.
- **Singularity image**[21]: Accessing hardware from Singularity container is somewhat different than for Docker. Singularity container sees the same hardware as the host system but runs under the user permissions (not root). For this reason, compiling a GPU-enabled Singularity container has to take these differences into consideration. Singularity image of Task 2 is based on Centos 7. At the compile time, exact versions of cuDNN and CUDA are installed. Environmental variables are set in order to install CUDA libraries during execution time. Specific Tensorflow and Keras libraries are also installed. Other Python dependencies are installed as well. Singularity containers have direct access to the user's home directory. Therefore, the code is not added at the compile time but instead, it is loaded during the execution. This is achieved by exposing Python interpreter as the container's default execution environment whereby the Python code can be passed as a parameter at the execution time. Similarly, Singularity container assumes that the data is present on a host path that is reachable by the container without mounting. The path of the input data is passed as a parameter during the execution.

The next step of this UC scenario implementation is to identify data services that need to be instantiated at the runtime in order to implement data management lifecycle. Architecture of micro-infrastructures used by LOBCDER implies that every UC have its own infrastructure composed of Docker containers hosted on Kubernetes cluster. Current containers being hosted for UC#1 are:

- **Staging service**: This is one of the core UC services. It is responsible for copying data between storages and HPC sites. Architecture of this service is composed of several Docker containers, which work together to coordinate data transfers. Figure 7

[20] The Docker file for this task is found at: https://github.com/recap/MicroInfrastructure/tree/master/ConatinaerAdaptors/uc1/stage-2

[21] The Singularity for this task is found at: https://github.com/recap/MicroInfrastructure/tree/master/ConatinaerAdaptors/uc1/stage-2

shows minimum setup for this service. The main entry point is a staging proxy, which takes in a JSON copy request job and figures out which of the adaptor containers can handle the job. This is achieved by analysing the job type and the source of the copied file. From this information, the proxy will query the service discovery container, which is hosted as a Redis server. The IP and port of the container adapter are retrieved and the job is forwarded to the container. The adapter runs a queueing system which queues copy requests and handles them in a FIFO order. The initial setup has an SCP adapter, which allows SSH request to remote data stores and starts SCP copy directly from the source host to the destination host. This relieves the data service from proxying data. Once the copying is done, the adapter notifies the proxy container and the container calls the webhook url defined in the job request.

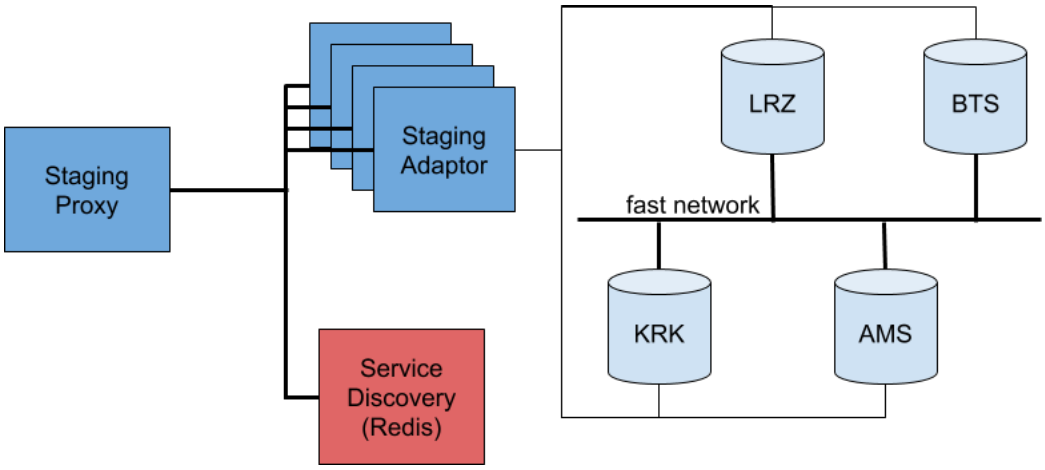

Figure 7: Staging service with multiple containers as adaptors

The job submission with the minimum information submitted is defined in JSON format and shown below. Amongst other information, job schema includes source, destination, type, and webhook. Access to the API is through tokens, which are generated by core-infra during booting of the container. The token is placed in the x-access-token header label. content-type is application/json

- **GET /api/v1/list** Get a list of all the files stored in all the storages.
- **Get /api/v1/find/:id** find the file named "id" and return its location/s.
- **POST /api/v1/copy** submit an array of copy requests to the staging service. E.g.

```
[{
    "id": "test123",
    "cmd": {
        "type": "copy",
        "subtype": "scp2scp",
        "src":{
            "type": "scp",
            "host": "SOURCE HOST",
            "user": "USER",
            "path": "FILE PATH"
        },
        "dst":{
            "type": "scp",
            "host": "DESTINATION HOST",
            "user": "USER",
```

```
            "path": "FILE PATH"
        },
        "webhook": {
            "method": "POST",
            "url": "WEBHOOK URL",
            "headers": {}
        },
        "options": {}
    }
}]
```

The code above submits a request to copy a file using SCP from one host to another. Credential management is managed in advance by core-infra during container booting, thus staging assumes credentials are in place. Many copy requests can be submitted at once (JSON array) and are processed separately. When a copy request is completed, the webhook is called.

- **GET /api/v1/status/:id** Get the status of a copy request with **id**.

- WebDAV service: For easy access by users, each micro-infrastructure exposes a WebDAV point, which allows user access to the storage.

5.2 Scenario 2: UC#2 development

The scenario we have derived from the LOFAR use case consists of two main steps: Data staging and calibration of the staged observation. The first step is responsible for communication with LOFAR Long Term Archive (LTA), where observations are stored, and for retrieving the data provided by the LTA. A single observation can be up to 16TB in size. The second step performs calibration on the observation and produces an image (or image set) of a patch of the sky to be analysed by astronomers.

- **Data Staging:** The data staging service runs as a Docker container. The Interactive Execution Environment provides the container with an observation identifier. When the data is moved to the cluster, the data staging service calls a webhook to let the IEE know that the data is ready. The code itself is written in Python. In order to find the correct observation files for a given observation, the LOFAR LTA client is used to query the LTA. This interface returns a list of SRM (Storage Resource Manager) URLs (SURLs). By using the LOFAR Python API to communicate with the LTA, the list of SURLs is staged. The progress of data staging is monitored periodically by polling through API. Once it is completed, the files can be individually downloaded by SRM. If possible, they are downloaded directly to the cluster. If SRM is not supported, an intermediate step is required. In the near future, LOFAR LTA is expected to switch to WebDAV. Thus it is possible that the data staging service will switch to WebDAV as well. In order to authenticate to LTA, a Grid proxy certificate is provided. The certificate is signed by a Grid certificate authority that is accepted by the LTA. Once the data is in the correct location, a webhook provided by the IEE is called.

- **Calibration:** In addition to the **target** sky patch the astronomer is interested in, a bright source within or close to that patch and for which "ground true" visibilities are known (termed **calibrator**) also needs to be specified. The observation files for both sources, provided by the LTA, are split into multiple subbands or frequency ranges,

usually 244 per observation. The calibration consists of two steps:

- **Direction independent calibration**: First, the calibrator is calibrated producing calibration solutions. The latter is then transferred into the target field, which is self-calibrated against a global sky model. This step can be performed by different LOFAR tools of which PREFACTOR tool built-in the *generic pipeline* framework is currently the most popular. Part of the processing can be done independently for each subband, making it embarrassingly parallel. However, other parts run sequentially. The final product of this step is a mapping matrix, which is applied to the entire observation. Low resolution FITS images with errors from ionospheric disturbances can be generated from this step. Currently, the whole step runs inside a Singularity container.
- **Direction dependent calibration**: This step starts where the precedent left off. It performs similar calibration for a number of directions within the LOFAR array beam. There are also different options from tool perspectives, but the most used one is the combination of KillMS and DDFacet making up the *ddf-pipeline*. This step currently runs on a single high-end node with at least 256GB of RAM and 3TB of scratch space. Research of whether this part can be split up over multiple nodes is still on-going. This step generates high-resolution FITS images from the target field where the errors from direction independent calibration are removed. This calibration step also runs inside a Singularity container.

5.3 Adaption of the PROCESS architecture

According to the development of the tasks for UC#1 and UC#2 described above, the initial PROCESS architecture (defined in the deliverable D4.1) was adapted (see Figure 8). The main improvement of the PROCESS architecture is the involvement of the micro-infrastructure approach, which is based on containerization. It was introduced in the deliverable D5.1 for the first time as a feature of the Alpha version of the PROCESS data services. Since the overhead of unnecessary software duplications caused by pre-processing tools (e.g. DISPEL) had been a serious issue, the approach allowing to every use case to have its own infrastructure was proposed. Currently, the micro-infrastructure is composed of Docker containers that are hosted in Kubernetes.

Micro-infrastructure is a very specialized and autonomous set of services and adaptors which interact across the extreme large data service-oriented infrastructure. Alongside the efficiency mentioned above, the approach supports scalability, high adaptability, modularity, and straightforward integration with the virtual layer. Since each use case has its own requirements and dependencies, modularity together with high adaptability are very important and useful properties of every exascale environment.

Another typical characteristic of the exascale environment is handling of different elements for processing, distribution, and management, which requires specific hardware, or nodes. These requests are possible to satisfy by the micro-infrastructure composed of dedicated nodes, or services addressing a particular request. Since the requirements are handled by virtualization typically (abstracting details of the hardware infrastructure, or the software stack), and so the micro-infrastructure offers a natural solution.

Figure 8 depicts the changes of the initial PROCESS architecture needed to involve the micro-infrastructure approach into the initial architecture. All the changes are highlighted in magenta. The main change is a new way of accessing data sources (through data adapters). The described approach also simplifies it. The new version has one “branch” instead of two “branches” (one dedicated to pre/post processing tools; e.g., DISPEL, and the

other dedicated to pure data access through the distributed virtual file system; e.g., LOBCDER). It also influences IEE (Jupyter is a part of micro-infrastructure, thus the IEE needs only a plugin for it), and LOBCDER (the data infrastructure management layer responsible for integration of lower adjacent tools was added).

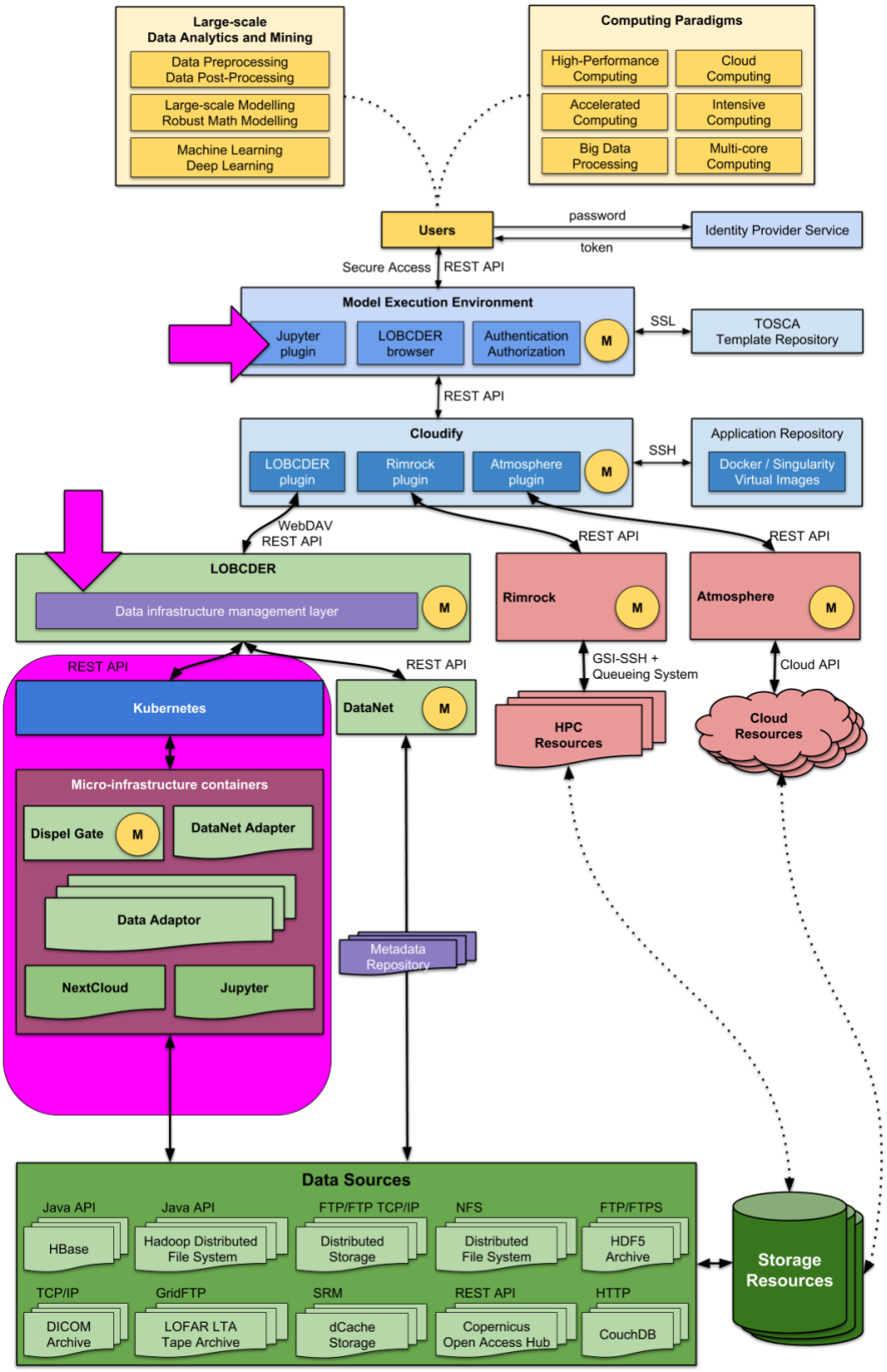

Figure 8: Updated PROCESS architecture.

The PROCESS architecture is also a result of applying the reference exascale architecture which represents the common features of the PROCESS platform (as well as every exascale-related platform, or application). On its top users are interacting with the platform through a secure access. IEE represents the environment for users, however, security is out of the project scope. Therefore, this aspect is not investigated anymore. The whole resource infrastructure is orchestrated by the virtual layer. The technology responsible for it is Cloudify. Below that layer is situated a virtual file system alongside HPC and Cloud managers. The virtual file system is containerized through micro-infrastructure. The main reason behind this decision is that the use cases have different requirements (e.g. need to access various data sources). Micro-infrastructure containers are managed by Kubernetes. Last but not least, HPC and Cloud managers. Both of them have to be scheduled and users have to have information coming from monitoring tools about their task as well as raw hardware infrastructure. Rimrock is used as a unified environment for managing HPC resources, and Atmosphere for managing of cloud resources.

5.4 Related deliverables

- D4.1 - Initial state of the art and requirements analysis, PROCESS architecture
- D5.1 - Design of a data infrastructure for extreme-large datasets
- D4.2 - Report on architecture evaluation and Dissemination
- D5.2 - Alpha release of the Data service

6 Conclusion and future work

The deliverable presents updated requirements analysis and updated PROCESS architecture together with validation of the first PROCESS platform prototype. All of the recommendations for the PROCESS architecture were applied on the initial PROCESS architecture and led to the second version - the updated PROCESS architecture. The second output related to the PROCESS architecture is the reference exascale architecture. This architecture describes a high-level structure (not as technically oriented as the PROCESS architecture) applicable for exascale scientific applications from a functional viewpoint. It provides principles for the design of a technical-level architecture.

The technical next milestone of the PROCESS project is the augmentation phase (MS4) in month 27. To reach the milestone, the results of the deliverable D4.3 will be extended as follows: The validation will continue to update the PROCESS platform (D8.1 in month 21) which will lead to the second prototype of the PROCESS platform (D6.2 in month 24). The prototype will use a chosen pilot application for a demonstration of its features. Afterwards, the deliverable D4.5 will report on a validation of the second PROCESS prototype platform in month 27. A development of the PROCESS architecture will also continue. The deliverable D4.4 will report its evaluation (in month 24) and consequently the next deliverable will describe the final version of the PROCESS architecture (D4.5 in month 27).

7 Appendices

7.1 Appendix A: Common requirements analysis (from D4.1)

Hardware requirements

- Access to HPC resources
- Access to Cloud resources
- Access to accelerated computing resources

- Access to external infrastructure from computing resources
- Access to data storage on the order of 1PB (distributed)

Software requirements

- Common tools for machine learning and deep learning
- Python development environment with support for Jupyter notebooks
- Java environment
- Apache Spark, Hadoop and HBase frameworks
- Support for containers such as Docker and Singularity
- Secure access to and extraction from external data resources
- Grid support
- Matlab environment
- Data extraction tools
- Large-scale modelling
- Predictive analytic methods
- Probabilistic risk calculation

All these requirements lead to different Execution Models, namely:

- Deep Learning
- Exascale Data Management
- Exascale Data Extraction
- Probabilistic Analysis
- Calibration
- Pre- and Post-Processing

In order to fulfil the requirements, one can derive the following building blocks as the starting point for the PROCESS architecture:

- Secure User Access to Management Platform
- Execution management Framework
- Distributed Data Management
    - Raw Data
    - Metadata
- Physical Data Storage and Archives
    - Storage
    - Archives
    - External Sources
- Computing Resource Management for at least (but not necessarily limited to)
    - High-Performance
    - Cloud
- Physical Computing Resources
    - High-Performance
    - High-Throughput
    - Cloud
    - Specialized accelerators (e.g. GPGPU)

To summarize, the following table overviews the generation of specific requirements (and thus building blocks) by use cases:

Table 3: PROCESS common concepts required per a use case.

PROCESS Execution Models	Deep Learning	Exascale Data Manage-ment	Calibration	Proba-bilistic Analysis	Exascale Data Extraction	Pre- and Post-Processing
Secure User Access to Management Platform	x	x	x	x	x	x
Execution Model Management Framework	x	x	x	x	x	x
Distributed Data Management						
Raw Data	x	x	x	x	x	x
Meta-Data	x	x	x	x	x	x
Physical Data Storages and Archives						
Storages	x	x	x	x	x	x
Archives						
External Sources		x		x	x	
Computing Management						
High-Performance	x	x	x	x	x	x
Cloud	x	x	x	x		x
Physical Computing Resources						
High-Performance	x				x	x
High-Throughput	x	x	x	x		
Cloud	x	x	x	x		x
Accelerated	x					

7.2 Appendix B: Common conceptual model (from D4.1)

Figure 9 shows a common conceptual model that is able to cover any of the requirements presented by the five PROCESS use cases:

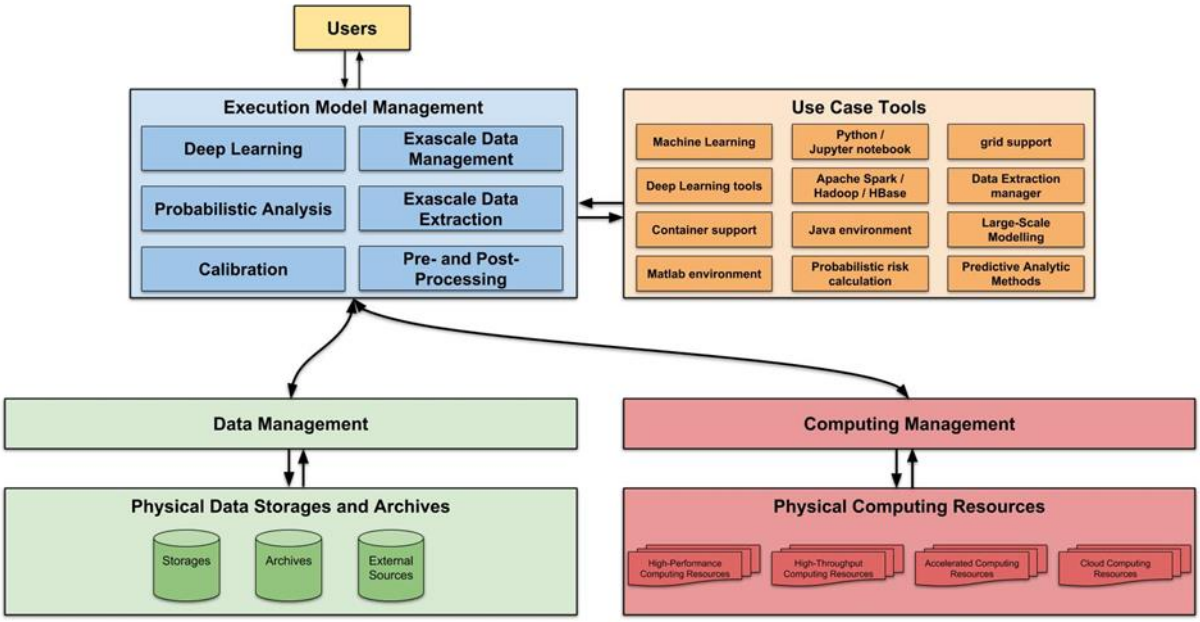

Figure 9: Common conceptual model.

The common conceptual model is enriched by certain (technical) necessities, namely:

- Secure access: As PROCESS is thought to serve several communities and use cases concurrently, a user and security management component is mandatory.
- Data Management: While a multitude of different data sources in different locations are accessed, a data management component is needed.
    - Access credentials to physical data stores must be passed on to the de-facto service provider and a potential translation must take place.
    - Potentially arising meta-data must be managed.
- Computing management: Similar to data management, access to a variety of physical compute resources, including credentials delegation, must be made available. Within this component a service multiplex and potential credential translation will be implemented and ensured.