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

| D4.2 | Report on architecture evaluation and dissemination | | |
|---|---|---|---|
| **Project:** | PROCESS H2020 – 777533 | **Start / Duration:** | 01 November 2017<br>36 Months |
| **Dissemination[1]:** | Public | **Nature[2]:** | **R** |
| **Due Date:** | 31 October 2018 | **Work Package:** | **WP 4** |
| **Filename[3]** | PROCESS_D4.2_Report_on_architecture_v1.0.docx | | |

## ABSTRACT

During the first year of the project, the PROCESS consortium led down the fundamental architectural components of the PROCESS architecture. Per design the targeted we aim at a highly modular infrastructure which can be easily adapted to various HPC and Cloud storage technologies and protocols. we choose to employ a divide and conquer approach whereby instead of building one monolith service infrastructure and hope it will scale to exascale, we implement a user/group micro-infrastructure approach. To facilitate the adoption and deployment of the PROCESS infrastructure. The proposed architecture uses container-centric virtualization where containers encapsulate atomic generic services and specific application codes which can be assembled in blocks to form various architectures to satisfy various application pipelines. The container-centric approach avoids several of the challenges related to intricate execution environment dependencies, that are often in conflict with other components of the application workflows.

This version is a draft of D4.2 and is under review.

[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] | Reggie, Adam | UvA | coordinator |
| **Contributing Authors**[5] | Habala, Ondrej; Tran, Viet | UISAV | Writers |
| | Meizner, Jan; Wilk, Bartosz | AGH | Writers |
| | | | |
| | | | |
| **Reviewer(s)**[6] | Höb, Maximilian | LMU | Reviewer |
| | Maassen, Jason | NLESC | Reviewer |
| | Heikkurinen, Matti | LMU | Reviewer |
| **Final review and approval** | | | |

## Document History

| Release | Date | Reasons for Change | Status[7] | Distribution |
|---|---|---|---|---|
| 0.0 | 15-06-2018 | Structure | Draft | |
| 0.1 | 14-09-2018 | Initial text Section 1, 2.1, 2.2, 2.3 | | |
| 0.2 | 21-09-2018 | Initial text Section 2.2, 3.1, 3.2, 3.3 | | |
| 0.3 | 24-09-2018 | Initial text Section 4.1, 4.2 | | |
| 0.4 | 28-09-2018 | Initial text intro, use case UC#1 | | |
| 0.5 | 17-10-2018 | Draft for review | In Review | |
| 1.0 | 31-10-2018 | Final Version | Released | |

This version is a draft of D4.2 and is under review.

[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

Deliverable D4.2 complements two earlier deliverables D4.1 (M6) and D5.1 (M9) by describing one possible implementation of the architecture presented in D5.1 and showing the preliminary evaluations of the different components composing this architecture. The objective of this deliverable is to connect all the pieces together to create the first PROCESS pilot architecture. The first pilot is not aiming at covering all the computing and storage resources available within PROCESS, but selecting a subset of resources representing sufficient and relevant challenge to demonstrate that the PROCESS data and computing services can be scaled out across sites and technologies (HPC and Cloud resources) with a minimal manual configuration. Selecting two sites that host a wide variety of relevant hardware solutions allows the project to maximise the technical challenges while at the same time minimising the time spent on administrative issues or liaising with the local support staff. Based on this rationale, the first pilot will target the resources available at two sites namely: Cyfronet in Krakow, Poland and SURFsara in Amsterdam, Netherlands. The first pilot focuses on building the PROCESS platform on HPC resources, hence choosing the two PROCESS HPC sites that have no scheduled upgrades or major maintenance during the reporting period was seen as an additional advantage. As a consequence, in D4.2 we focus more on discussing and presenting the PROCESS services which enable to use HPC resources. However, we also present the PROCESS components that are currently in development and will support the integration of Cloud resources.

In D4.2 we present the progress achieved so far to port and prepare the PROCESS use cases to the PROCESS infrastructure. During the last three months, use case owners and infrastructure developers worked together to test and evaluate the various components of the PROCESS infrastructure using a set of pilot use cases (simplified versions of the actual PROCESS use cases). Within this deliverable, we present the progress in porting use cases UC#1, UC#2, and UC#4, and report the progress on UC#3 and UC#5. The former group of the use cases are based on mature, specific usage scenarios that include non-trivial capacity and capability requirements. For this reason, they represent ideal pilot cases for evaluating the architecture through deployment on the PROCESS infrastructure.

In D4.2 we evaluate the core elements of the architecture independently. The overall evaluation and benchmarking will be covered in another deliverable. As for the dissemination, we report on the ongoing effort to create a software repository, where all the software will be made available for users outside PROCESS. We also describe a strategy to publish the APIs following openAPI specification.

# List of Figures

# List of Tables

# 1 Introduction

At this stage of the project, all the components composing PROCESS architecture are being developed as standalone components. As a preparation for the first milestone of the project planned in M12, we decided to follow an approach for the evaluation of the architecture, which gives the developers a maximum flexibility (avoiding whenever possible unnecessary interdependencies as the progress of the different components is not the same) while still ensuring an easy and seamless integration, when comes the times for the first *alpha release* of the PROCESS platform. We follow a three-step evaluation method:

- **Technology and performance analysis of standalone components**: Technologies used for each component in PROCESS are evaluated and tested separately.
- **Applicability to the PROCESS use cases**: all components will be checked against the application requirements identified in Deliverable D4.1 and D5.1.
- **Definition of the application programming interface**: all components should have a clear and well-documented API, following best practice in API documentation that define exactly the functional behaviour of the component.

The structure of D4.2 reflects this evaluation approach: each component composing the architecture is described in a separate section namely:

- Data services (Section 2),
- Computing services (Section 3),
- Service orchestration and User Interfaces (section 4).

Each section is further divided into four main sub-sections: The implementation, evaluation and limitations, applicability to use cases, and finally the API subsection.

## 1.1 Pilot architecture

The initial architecture, detailed in D4.1, partitions the architecture into 3 main parts; user interaction virtualization layer, data layer and compute layer. A user accesses the architecture through an application-oriented scientific gateway, which provides a secure access in the form of an interactive environment represented as a script Application Programming Interface (API), command-line interface (CLI), and graphical user interface (GUI). Authentication and authorisation of the user is performed by a dedicated service. In the initial architecture, it was envisioned that applications will be deployed in containers or virtual machines. Container centric approach avoids several of the challenges related to intricate execution environment dependencies, that are often in conflict with other components of the application workflows. The approach will also harness the full potential of the infrastructure through secure environments that are separated from each other. However, it means virtualization/containerisation of an application environment as well as its execution environment. Deployment of the application services is done through a virtualization layer which is responsible for the resource allocation, the configuration of all services, and the management of their whole life-cycles.

In D4.2 we start implementing the architecture as defined in D4.1. The approach we employ is that the major parts of the architecture are developed independently while describing APIs for eventual integration. One of the main goals of this pilot architecture is to expose implementation obstacles into how the main parts of the architecture will work together as well as obstacles and limitations of each sub-architecture. These obstacles will drive the iteration of updating the architecture in D4.3.

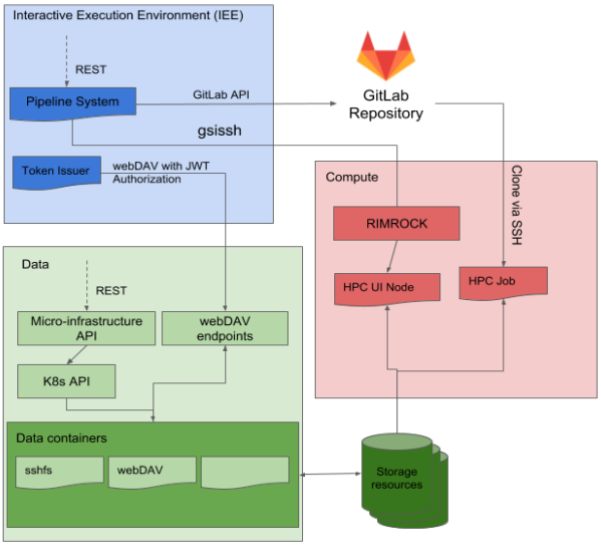



*Figure 1: Pilot architecture*

Figure 1 depicts the pilot architecture with the main three loosely coupled components and the interaction between these components.

**The data services** expose two interaction points; a REST management API where users can manage their private micro-infrastructure and a set of external infrastructure endpoints such as WebDAV. The authorization to the web services is ensured through tokens. Generation of the access tokens is achieved through a global access and authorization service.

**The compute services** are composed of the pipeline system embedded into the Interactive Execution Environment (IEE). It acts also as the main UI gateway to the platform as well as the RIMROCK component responsible for direct interaction with the HPC clusters via their UI nodes. The UI node access is achieved through the gsissh - extension of the SSH protocol allowing usage of the GSI Proxy Certificates (X.509) for credential delegation (instead of typically used long-lived key pairs or passphrases for SSH). Both IEE, as well as job scheduling via RIMROCK, communicate with the Gitlab component - the IEE to enable selection of the appropriate code version (such as tag or branch) and the job scheduling to run the selected workload. Inputs for the computation may be uploaded via the IEE to the data service. Also, when computations are finished outputs may be fetched from the data service using the same mechanism.

**The orchestration service** helps, automatically or on user requests to deploy dynamically on demand computations to cloud systems. This includes virtual execution environments and data processing services. The services need to be described using TOSCA templates that specify service topologies including hardware and software environments. Based on the templates, Cloudify, the orchestrator, will deploy the services and give users access to them

## 1.2 Pilot use cases

The five PROCESS use cases are progressing at different paces (this progress is reported in detail in D2.1). In deliverable D4.2, we discuss the applicability of each component composing the PROCESS architecture to each use cases. More specifically, we cover how the different PROCESS use cases will scale out across geographically distributed computing sites and how they access data in such a distributed environment. The PROCESS use cases have different

data access patterns based on data stored on third-party storage resources for UC#1, UC#2, UC#3, UC#5 or data streams in the case of UC#4. The use cases also have datasets stored in various technology ranging tape-drive for UC#2, to datastore accessed through SpringData[8] for UC#4. Because of this multitude of data access patterns and storage technologies, we aim at a programmable infrastructure which can be easily adapted to various HPC and Cloud storage technologies and protocols. With the proposed approach we will be able to build a virtual infrastructure dedicated to each use case in PROCESS. To ease the deployment of the use case data processing pipeline, the application code is containerized. Because most, if not all, HPC centres adopted Singularity as container virtualization layer for security reasons, all the use case code which has adopted a different container technology (Dockers) had to be converted in to singularity, this step is not always straight forward due to various reasons such as no root privileges and 3rd party images specific to certain container technology. For instance, in the case of UC#1, which uses NVIDIA GPU Cloud (NGC) Docker images for deep learning and data analytics optimized for GPUs, the transition to Singularity turned out to be more time consuming. For use case UC#2, the prefactor-CWL pipeline implementation described in D2.1 uses both Docker and Singularity containers.

The use case developers will access both storage and computing resources through a simple and easy to use portal namely the Interactive Execution Environment (IEE) described in Section 3 - page 23, which allows them to create the various use case data processing pipelines using HPC resources, target of the first pilot, and at later stage also cloud resources using Cloudify as described in Section 4 - page 27.

# 2 Data services

## 2.1 Architecture

Data is bound by physical constraints which can be loosely described as the capacity to store, address, and transfer. These physical limitations can be somewhat attributed to one of the V's of data[9], that is Volume and are shaping the data architecture on the Internet. As happened with computer architectures, several data layers were implemented as a mean to hide latency. Each layer further away from compute becoming progressively slower but larger in capacity. In a way, this organization of data access can also be seen happening on the Internet. For example, tape drives offer the largest capacity but are very slow in access time while HPC file systems offer high throughput with relatively low capacity. Figure 2 depicts the data organization on the Internet. Our work focuses on the L2 layer where we federate data across computing sites in an effort to increase throughput and data capacity of scientific applications.

[8] https://spring.io/projects/spring-data

[9] The four elementary characteristics of big data known as the "v's of big data" namely : Volume, Variety, Velocity, and Variability are defined in the NIST Big Data Interoperability Framework: Volume 1https://bigdatawg.nist.gov/_uploadfiles/NIST.SP.1500-1.pdf

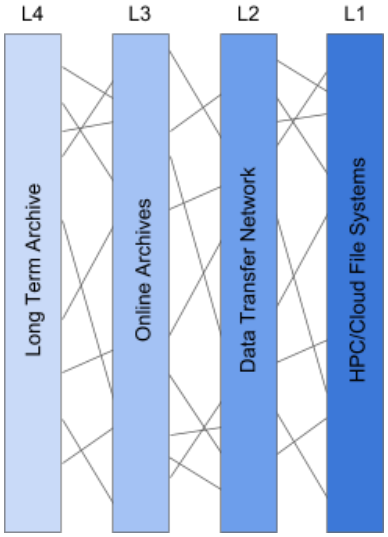



*Figure 2: Internet Data Layers*

In D5.1 we have presented different implementations of the Data management sub-component of the initial PROCESS architecture described in Deliverable D4.1 page 54. Among these solutions, we have decided to implement the PROCESS Data Federated Management Systems (PROCESS-VFS) around well-established storage federation tools like nextCloud. The choice to start with nextCloud is motivated by the fast-growing adoption of nextCloud, which is at this moment taking over other similar approaches. Also, the new features of the nextCloud 12 “Global Scale Architecture” make it a very promising technology to use in PROCESS to reach the exascale target. However, Figure 3 shows the pilot architecture we are currently considering, for now, the pilot is limited to two PROCESS sites (Amsterdam and Krakow), but by design, it is meant to be extensible to other sites within PROCESS and later on beyond the project sites.

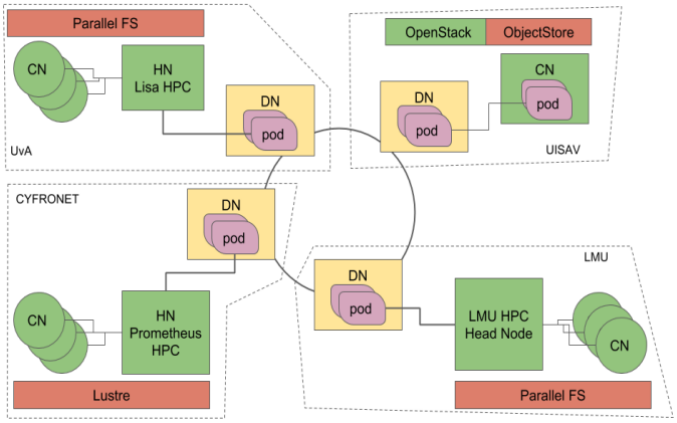



*Figure 3: Data Nodes (DN) form a distributed programmable layer that implements data federation amongst HPC site. HN: Head Node, CN: Compute Node*

The focus of the first pilot test bed for data is to achieve data storage, management and transfer capabilities between HPC sites. The basic data architecture of many HPC sites is a high-performance parallel file system such as Lustre[10] and Ceph[11]. Remote access to these file systems is usually limited to ssh[12] based methods and GridFTP. On the other hand, cloud-

[10] http://lustre.org
[11] https://ceph.com
[12] Secure Shell (SSH)

based HP systems can be easily programmed which means data is not restricted to data services provided by HPC sites. In PROCESS we have a mixture of HPC and Cloud-based systems thus we plan a highly programmable infrastructure which can be easily adapted to HPC and Cloud alike. To make sure that our initial pilot is flexible and scalable, e.g. easy to extend to multiple datacentres and malleable to different application data needs, we choose to employ a divide and conquer approach whereby instead of building one monolith data service infrastructure and hope it will scale to exascale, we implement a user/group micro-infrastructure approach. With this approach, users build a virtual infrastructure dedicated to their application needs e.g. different data access, caching and security. The architecture uses container-centric virtualization where containers encapsulate atomic generic data services and specific application codes which can be assembled in blocks to form various data architectures to satisfy various application pipelines. The length, complexity, and data access patterns of such pipelines can vary based on the type of workflow. E.g. applications with large staging data might require caching mechanisms to employ latency hiding while other applications might not require such capabilities. Through this micro- infrastructure approach, the data layer provides a programmable layer where specific data functionality which eases the burden of data transfers (e.g. transcoding) can be programmed into this layer.

The architecture of our proposed data service is a virtual distributed programmable layer. The main criteria for such a layer are to be scalable, distributed, adaptable and secure with respect to data. With regard to scalability, we require that our approach needs to deal with large amounts of data. The distributed feature allows managing data with location restriction and optimizes compute routines based on data locality. The ever-increasing complexity of applications means our system needs to be adaptable now and for future unknown applications. Security and data ethics are increasingly becoming a major issue in scientific application especially applications from the medical sciences thus a data service must address this issue. With this attribute in mind, we will discuss in the next section our primary implementation approach to cater for these attributes.

## 2.2 Implementation

Implementing a system that is scalable, distributed, adaptable and secure is a massive undertaking. To facilitate our implementation and satisfy these attributes we make use of various tried and tested technologies. Our implementation revolves around Kubernetes (K8s) as our main technology. Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications[13]. K8s enables hyper-converged infrastructures[14] (HCI). This allows maximum dynamicity in provisioning infrastructures e.g. a compute and data cluster can be provisioned from globally distributed nodes whereby virtualized networking creates a LAN between nodes and virtualized data.

K8s gives us the means to create the foundations of a programmable layer for our data. Without delving too deep into the architecture of K8s, it is sufficient to describe a few concepts that are central to our approach. K8s manages container deployment in a master-slave approach. A minimal K8s cluster is composed of a master node and a number of worker nodes onto with containers are scheduled. K8s introduces some abstract concepts that allow logical organization of the virtual infrastructure. The minimum scheduling unit in K8s is a pod. A pod is a group of containers which, when scheduled share the IP, thus two containers in the same

[13] https://kubernetes.io

[14] Hyper-convergence is a software defined infrastructure where all elements of the traditional hardware infrastructure (compute, data and network) are virtualized s

pod can talk to each through localhost. Pods are somewhat the equivalent abstract representation of a virtual machine while the containers are the abstract representation of services within the machine. Multiple pods can be further grouped together to form a network where pods can communicate with each other through IPs or domain names. The latter being managed by K8s and a preferred way to discover pod IPs since service IPs can change (e.g. crashing and auto restarting). Data in K8s is managed through different data backends. A popular data backend is Ceph which itself manages distributed block storage and K8s can mount such Ceph blocks. Similarly, networking is virtualized and pluggable. Network plugins follow a Common Network Interface (CNI) whereby networking capabilities can be provided by the different plugins. Common CNI plugins are Calicio[15], Cilium[16], Flannel[17] and Weave[18]. Every network plugin focuses on different criteria, e.g. Cilium provides Layer 7 policy enforcement between pods.

Through the capabilities of K8s, we can realize our micro-infrastructure approach to data management on a per user/VO or application bases. Containers offer the building blocks of the data infrastructure services. The building blocks of a micro-infrastructure are four main types of containers.

- **Adaptor containers**: Adaptor containers provide the required functionality to connect remote data stores. This is especially the case with HPC. Typical HPC implements a high-performance parallel file system for fast access to compute within the cluster. Remote access to the parallel file system is usually limited through ssh and GridFTP. To create a federation of data across different HPC sites, our layer needs to interface into the different sites using the different mechanisms and manage user credentials to access such file systems. For this reason, we have the notion of adaptor containers which encapsulate access to a particular site including handling access secrets and expose a standard interface, e.g. WebDAV to expose a data storage internally to the micro-infrastructure. In our pilot implementation, we use 2 such adaptor containers that connect Prometheus HPC in Krakow and Lisa HPC in Amsterdam through standard ssh protocol. The adaptors each expose a WebDAV point internally so that other containers can connect to.
- **Data containers**: Data can also be stored directly in the micro-infrastructure layer. This allows for a hybrid HPC-cloud approach to manage data. Reasons to store data directly in this layer can vary on application bases. For example, the layer is used as a primary data store where the data is fully managed in this layer. Intermediate data, where distributed workflows across HPC sites need to communicate data.
- **Interface containers**: Interface containers provide an external interface into the infrastructure. Interface containers provide the user interface to manage and manipulate data while also provide interfaces to external compute programs such as HPC clusters. In our current pilot approach, we provide a WebDAV external interface to access the internal infrastructure. WebDAV is a widely used interface that allows users and programs to mount remote storages to access files from remote storages.
- **Logic containers**: The complexity of application deployments and their data access patterns means specific applications will need a specific way to deal with data e.g. streaming or file based. Other routines could optimize data transfers e.g. caching, where

[15] https://docs.projectcalico.org/v1.5/getting-started/docker/tutorials/basic
[16] https://cilium.io
[17] https://coreos.com/flannel/docs/latest/
[18] https://www.weave.works

multiple HPC centres pull from the same datasets and caching will minimize such transfers. As datasets grow, data transfer/transcoding scheduling will also become an important feature whereby a schedule is used to plan ahead data staging for applications.

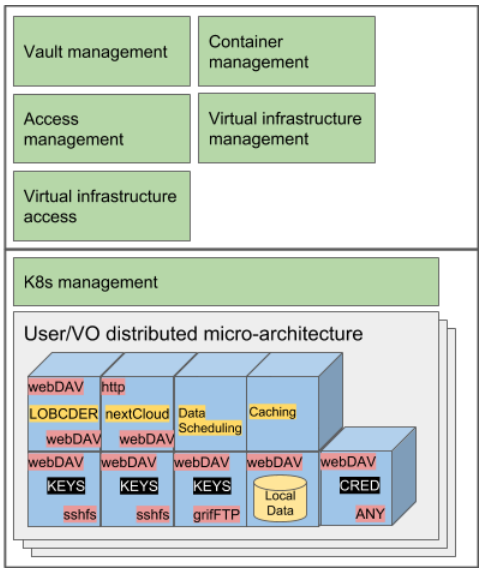



*Figure 4: Data services micro-architecture approach*

Apart from the container building blocks, supporting REST services are also needed to manage the lifecycle of the micro-infrastructure:

- **Vault management**: Adaptor containers need access credentials to be able to connect to remote storages. The vault management service handles the user's credentials securely and provides the required credentials for the adaptor container.
- **Container management**: Creates and manages the building blocks for the micro-infrastructure.
- **Virtual infrastructure management**: Manages the lifecycle of a micro-infrastructure e.g. composing, initializing, stopping.
- **Access management**: Manages access and credential integration with other PROCESS services.
- **Virtual-infrastructure**: Manages external access to the micro-infrastructure e.g. maintaining top-level WebDAV endpoint. This service generates K8s infrastructure recipes by combining different containers into one or many pods described in YAML DSL.

Although such an architecture provides the capability to access data globally, yet we have not tackled optimized data transfers. Having WebDAV as a standard interface between containers and external interface means that transferring files from one site to the next entails that data must pass through the containers themselves. This is certainly not optimal thus we are developing a WebDAV server that can handle command delegation so that data transfers happen with the least amount of server hops. E.g. transferring data from Prometheus to Lisa could be done directly from either HPC head node. By hooking into HTTP methods such as COPY which is used by WebDAV, we can percolate the copy command down to the container adaptor which in-turn would know best how the facilitate a copy. In the example of using sshfs on Prometheus and Lisa, a ssh copy command would be called directly on one of the head nodes thus the data would not pass through unnecessary nodes.

## 2.3 Evaluation and limitations

### 2.3.1 Experiments with container technologies

In these preliminary experiments, we targeted well-known container-based virtualization, namely Docker and Singularity, mechanisms that are currently adopted in industry-wide and scientific computing communities. We have found that both technologies share some features with each other but differ when investigating their underlying design and in some characteristics, the mechanisms differ in the interaction and dependency between the container engine and host machine. Both technologies approach the needed virtualization from different perspectives; Docker focuses more on adopting a Development and Systems Operation (DevOps) philosophy where communities have embraced it as a de-facto standard format for micro-service type deployments (one service per container). Whereby Singularity harnesses the portability of containers to bring the mobility of computing in the HPC community to satisfy the requirements of achieving scientific computational usage in cluster environments and computational centres. The different angles narrow down to the requirements of both technologies in different communities, that is for industry usage and scientific world respectively. We have also discovered multiple aspects of both container systems that bring along their own set of limitations or design flaws, with this in mind; containers are pushing the frontier of virtualization techniques where resource, filesystem, process, and network isolation is of high concern. Both container mechanisms have proven to be potential solutions when reproducibility and portability of applications are required.

As our programmable virtual data layer architecture is composed of distributed container technologies, we wanted to investigate further the impact and overhead introduced by such containers. To gather performance metrics, we employed a typical pipeline application composed of data ingestion and processing (Figure 5). Each pipeline stage is encapsulated into a container (Docker and Singularity) and execution and communication are coordinated between containers. Input and intermediate data in our setup allow users to read and write from a wide range of data sources, file systems (LFS, HDFS, S3, EXT3, EXT4 and so forth) and web services. Storage is decoupled from the component execution environment such that each component can ingest data and output artefacts into a shared mounted storage which is mounted in the container at runtime execution.

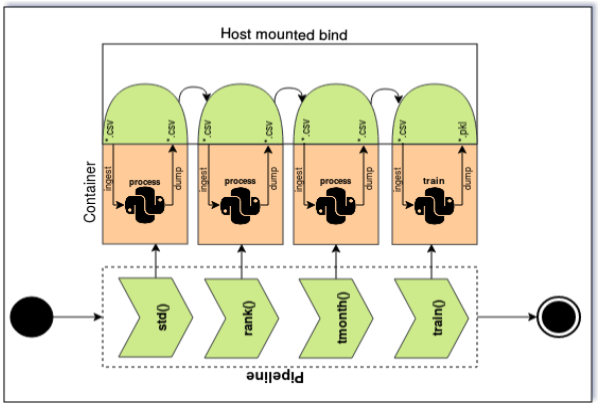
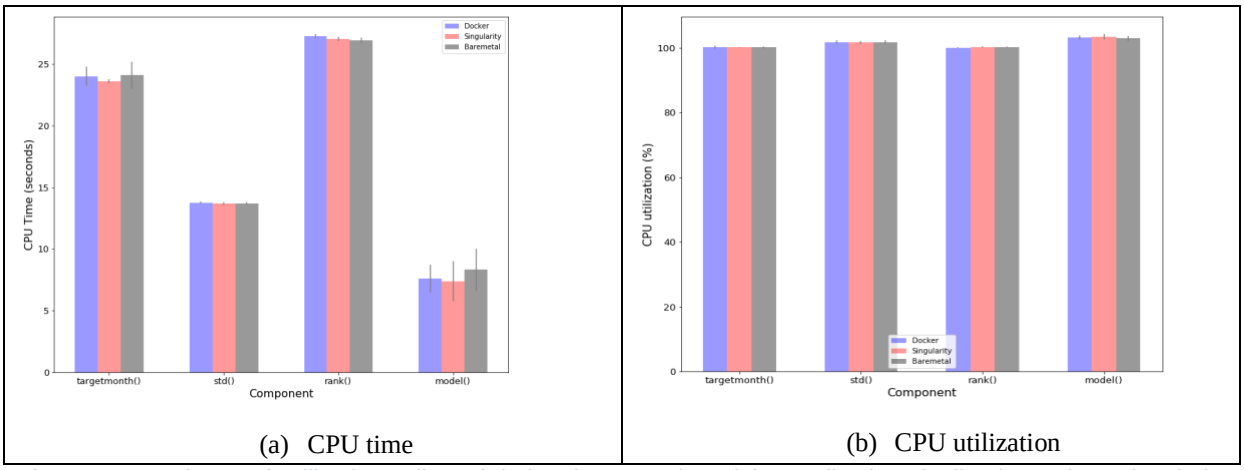



*Figure 5: main functionality of the application workflow which consists of three pre-processing steps and one training step*

In order to determine the applicability of our solution, we have laid out a number of experiments that allow us to measure the performance of such a workflow. By doing so, we consider factors such as CPU usage, resident set size during its lifetime, context-switches and other resource

indicators for each component in the workflow and compare these values with a system that is not virtualized (bare metal).

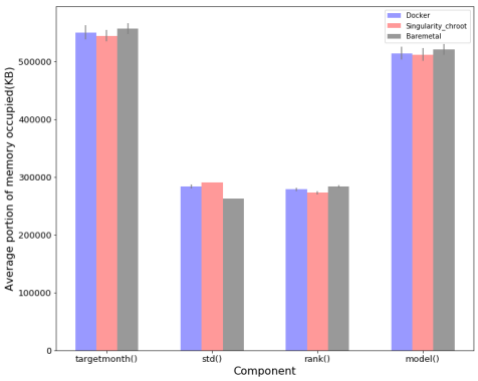
The first performance metric, we measured is the average CPU time for each component composing the application pipeline. We observe the results across the experiments when using the Docker-based or Singularity-based pipeline show near or even better than native timings. Here we measure the user CPU-time for our running components, where for each component initiated we measure the duration the CPU was utilized for performing the computations and processing required. The CPU-time allows us to quantify the amount of time the system was actively working with the input as well as the amount of processing power delay acquired when achieving the desired outcome of a process which in our case is specified per component analysis.



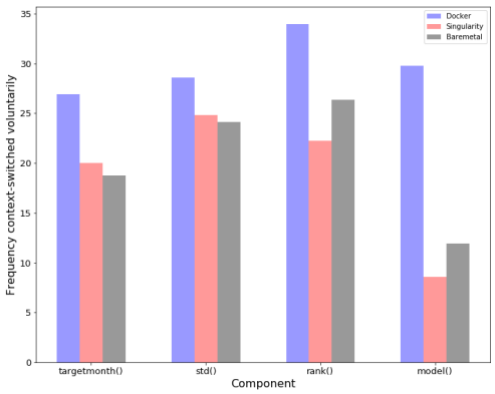

(a) CPU time (b) CPU utilization

*Figure 6: CPU time and Utilization collected during the execution of the application pipeline in Docker, Singularity containers and the bare metal execution*

As a second performance metric, we measure the resident set size expressed in KB for each component in the application use case. The Resident Set Size (RSS) represents the maximum amount of main memory a process, or in our case component, has occupied in real RAM (not swapped) over its lifetime (where less is better). This accounts for the code, shared libraries, data and other memory types that are involved when executing a specific component. This metric allows us to determine the amount of memory intake each component has consumed considering the similar layout across environments. Furthermore, the memory usage metric collected serve as an indicator of the memory deviation when compared to the bare-metal environment, here we investigate whether components occupies larger parts of the RAM when utilizing a container-based mechanism. This metric is useful especially in machine learning; this is due to the reasonable amount of memory a machine requires to be able to handle large amounts of information.




(a) Memory intake




(b) Context-switches

*Figure 7: Memory intake and context-switches registered during the execution of the application pipeline in Docker, Singularity containers and the bare metal execution*

Finally, we measure the frequency of context switches per component in application pipeline (Figure 7b). When managing multiple processes, an operating system switches context from one process to another, where a process here is defined as an executing instance of a program. Context switching is a relatively heavy task since it involves the operating system existing user space and entering kernel space and vice-versa. Thus, too much context switching in a system can be detrimental to the throughput of the whole system. In Figure 7b we plot the voluntary (explicitly changing process thread) context-switches from the first use case, as seen the values extracted here highly vary from each other. Here the higher the frequency of context switches we observe the more computationally intensive that specific operation will be. In hindsight, we remark the highest frequency of context changes in the Docker-based workflow than the other two environments. Operations executed in the singularity-based environment made overall less frequent usage of context changes for the first two components while for the last two, the bare-metal environment achieved the least frequent context changes.

### 2.3.2 Experiments with federated data access

One of the core elements of the PROCESS data service is the federated data access service. As explained in Deliverable 5.1 Section 2 - pages 8-29. The underlying fundamentals of federated data are the ability to move data between locations (Table 1). This is ultimately limited by the physical networking limitations and protocols being used. For this reason, we first investigate the inter-site bandwidth matrix.

*Table 1: storage resources available in PROCESS*

| Storage available for the pilot | Access protocols | Authentication mechanisms |
|---|---|---|
| **CYF (20TB, type)** | sshfs | password/rsa keys |
| **UVA (size, type)** | sshfs | password/rsa keys |
| **UISAV (30 TB, HDD RAID via NFS)** | Sshfs, FTPS (not SFTP) | RSA keys (SSH v2) |
| **LMU (100 TB, Data Science Storage)** | GridFTP, NFS, sshfs | NFS exports, x509, RSA keys |

*Table 2: Upload and Download speeds for copying a 1GB file between HPC sites*

| Up/Down - Protocol MB/s | pro.cyfronet.pl | lisa.surfsara.nl | lisa-gpu.surfsara.nl |
|---|---|---|---|
| pro.cyfronet.pl | - | 53.4 / 55 - SCP | |
| lisa.surfsara.nl | 50.5 / 30.3 - SCP | - | Shared FS |
| lisa-gpu.surfsara.nl | 40 / 8.9 - SCP | Shared FS | - |

From Table 2 it is immediately evident that although the HPC sites are connected over GEANT network, the available speeds and protocols between these sites is very limited. Such limitations will severely influence the distribution of applications to multiple sites since data management and placement at this level is crucial to the overall performance of applications. The bottleneck in data movement further emphasizes the need for smart data layer where certain functionality such as transcoding of data is done in the network thus reducing the amount of data that needs to be shipped. Such functions can be seen as yet another container in the application's micro-infrastructure.

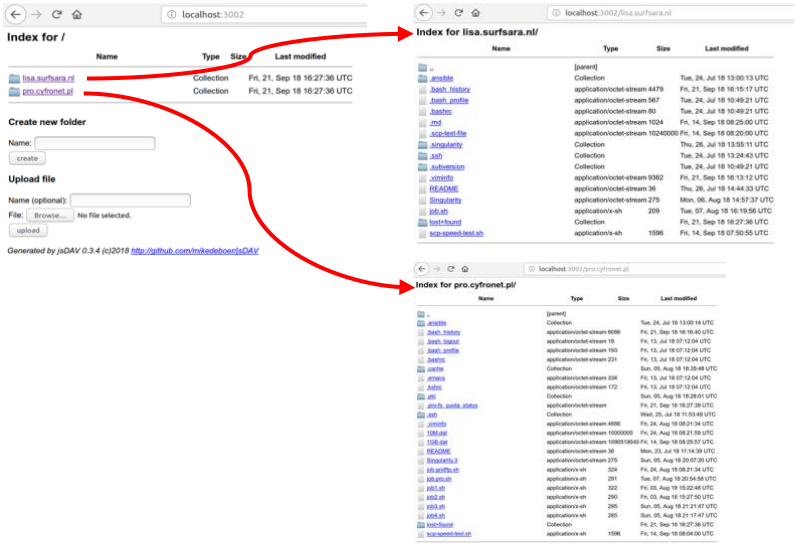



*Figure 8: Micro-infrastructure federating access to Prometheus HPC and Lisa HPC*

We tested an initial micro-infrastructure composed of 3 container blocks (Figure 8). Two adaptor containers which connect through sshfs to Prometheus and Lisa and another interface container which exposes the micro-architecture as a WebDAV endpoint. The micro-infrastructure is described in a K8s YAML file which allows for parameterization of the infrastructure so that usernames, hosts, folders and ssh keys can be parameterized. Sensitive data such as keys are handled separately in K8s. Secret volumes are created with sensitive data which are mounted when deploying a pod. This way no sensitive data is exposed in the micro-infrastructure description file.

### 2.3.3 Experiments with advanced pre-processing of data

One technology to be used in WP5 in PROCESS is the DISPEL Gate technology[19]. This is a mature technology (TRL 7-8) created in a previous European research project (ADMIRE, FP7). During the ADMIRE project, numerous experiments with the technology have been performed,

[19] DISPEL Gate description, deliverable D4.1. H2020 PROCESS project, 2018.

on more than 10 pilot applications. We have re-evaluated DISPEL service in the light of the application requirements collected in the PROCESS project. The DISPEL technology itself is readily available as a VM image and can be easily containerized to be in line with the approach adopted in PROCESS. The re-evaluation of DISPEL has been centred around the properties that are necessary for the PROCESS project namely: the scalability, fault tolerance, and the minimisation of cross-dependencies:

- **Scalability** - it is possible to deploy numerous cooperating DISPEL Gateways. Each Gateway is able to operate independently of the others, but if necessary, it can delegate the data process to another DISPEL Gateway. This feature is very interesting from the exascale point of view, as it enables DISPEL Gateways to keep up with the increase of the load.
- **Fault tolerance** - not only can Gateways operate independently - within the domain covered by the services and data resources the Gateway manages - but by deploying several duplicate Gateways, with duplicate sets of data services, the duplicate services can kick-off at any time the original services fail or become unavailable.
  - **Minimization of cross-domain dependences** - DISPEL, the high-level data process language, has been designed with the “division of responsibilities” in mind. Such an approach allows creating complex data processes by several domain experts - distributed computing experts, data mining experts, application domain experts - without these groups of experts having to gain expertise outside their domain. This allows
  - distributing computing experts to set up the infrastructure, without having to know data mining or the details of the applications to be run on the infrastructure;
  - data mining experts to prepare data processing services and describe their usage, without having to know how to deploy the distributed computing infrastructure, or how to provide data and input parameters to the data processes;
  - application experts to run applications in their domain, using pre-prepared distributed infrastructure and high-level data process descriptions, just by providing data and configuration for the data processes.

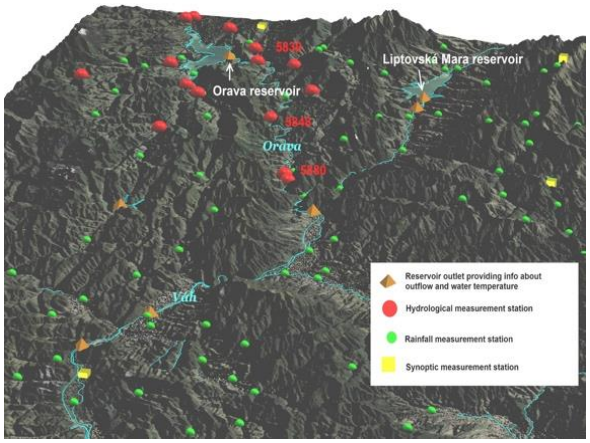



*Figure 9: data acquisition points of the ORAVA experiment*

Here we describe some experiments we have performed while re-evaluating DISPEL technology.

We used the ORAVA example which aims to predict water level and water temperature in the river ORAVA (Northern Slovakia), used to predict the possibility of local flooding caused by the build-up of ice on the River during the first thaw in early spring. We can regard this example as a very simplified version of the PROCESS UC#3, ORAVA example is a relevant local hazard. The scenario uses input data from three main sources:

- Hydrological measurement stations (red bubbles in Figure 9)
- Weather predictions (gridded data)
- ORAVA Reservoir measurements (water level and temperature at the reservoir outlet - brown polyhedron )

These data are first integrated and then mined for a prediction of water temperature and water level at the measurement station points along the river.

To simulate multi-input data sources data processing pipelines (which is a common data access pattern in PROCESS), we have distributed the data among three separate sources, each accessible by a DISPEL Gate. For container and service deployment we replaced the original Grid and OGSA-DAI technologies used in ADMIRE with the PROCESS technologies developed in JRA4: Service Orchestration and User Interfaces (4 - page - 27). We set the target to reproduce the results obtained in the ADMIRE project[20] using the PROCESS approach, to achieve this goal, we kept the high-level data process the same as in the original experiment.

## 2.4 Applicability to use-cases

The micro-infrastructure approach will express its full potential when applications are federated to multiple computing platforms. In such a scenario, data scheduling, caching, filtering, and transcoding will optimize the data movements between sites. For applications with enough data parallelism, this will provide an added scale-out feature.

At this moment the use cases (UC#1 and UC#2) are being containerized either using Docker or Singularity, which are composed to create various versions of the use case data pipeline process. This container centric approach is not only useful for the use case developer as it enables them to easily modify/extend their data processing pipeline to test new algorithms, approaches, etc. It is also useful for the infrastructure developer as containers are in principle easy to deploy and run on different systems. The Experiments with containers described in Section 2.3.1. were performed on applications which are very similar to the PROCESS use cases. In terms of overhead, we expect that results, shown in Section 2.3.1, to give a reasonable estimation of the overhead when the real PROCESS use cases will be ported to the first Alpha release of the PROCESS platform.

**UC#1** aims to use artificial neural networks (ANN), the types of parallelism employed by the neural networks are of three types; model, pipeline and data parallelism. The nature of an ANN is that communication is paramount for updating weights on every iteration. The easiest to implement is data parallelism where multiple examples of data mini batches are distributed to different nodes which can work in parallel and at the end of an iteration parameters are updated through an all-reduce function. An all-reduce is a heavy-weight function on the network thus with limited interconnects between computing sites, a single ANN might not scale since communication will become a bottleneck. A different approach is to train multiple ANNs on different sites for example to optimize hyper-parameters. In this case datasets, hyper-parameters and algorithms are distributed to different sites to train in parallel, the output of which is compared and used for the final ANN. In this case, the micro-infrastructure can coordinate the data outputs between different sites.

**UC#2** has opportunities for multi-site distribution since the first part of the pipeline - mainly direction independent calibration - is embarrassingly parallel on data. A 16TB observation is split into 244 1.2GB chunks representing different wavelengths. These chunks can be

[20] Habala, Ondrej - Šeleng, Martin - Tran, Viet - Šimo, Branislav - Hluchý, Ladislav. Mining environmental data in ADMIRE project using new advanced methods and tools. In International Journal of Distributed Systems and Technologies, 2010, vol. 1, no. 4, p. 1-13. ISSN 1947-3532.

processed in parallel on 244 nodes after which a merging operation takes all outputs from the 244 chunks. For a typical LOFAR observation, this step takes about four hours. One way to achieve scale-out on multiple-sites is to have the data micro-infrastructure for this application to distribute the chunks to different sites. As one can notice the throughput of the pipeline is limited by the slowest node processing a chunk thus the data scheduler can employ some smart logic to distribute workloads so that all chunks are processed in approximately the same time. Adding to this, UC#2 has more complex data staging in PROCESS since data is staged in from tape-drives which means the data micro-infrastructure will employ some logic containers that can help automate this process.

Direction dependent calibration is the next step in data reduction. This presently takes four days on a single high-end CPU with 256 GB RAM.

**UC#3:** While this use case does not have specific requirements regarding the architecture, it fits the generic data processing pipeline envisaged in PROCESS. Since this use case is targeting “long tail of Science”, the datasets are relatively small, the existing pre-processing solutions (creation of archives of multi-file directories on-demand) are sufficient for the current, known use cases. The metadata used in UC#3 is based on a simple directory structure. However, from the usability point of view, it is possible that the UC#3 datasets will be used by individuals and communities that have no traditions for designing and using complex workflows. Thus UC#3 will inform the project’s approaches related to debugging tools, interactive help systems and managing license agreements (including click-through ones) in a way that is both intuitive and efficient. With the size of the current dataset (~1.5TB), scalability is not perceived as a major issue. The expected increase of the datasets is about 1 TB/year, however in case there is a dramatic increase in the dataset traditional website load-balancing methods will be enough to keep with the growth of the data, the container centric approach promoted in PROCESS is also applicable to achieve scalability in UC#3. For the high-resolution modelling pilot, described in D2.1, the computation is still expected to be easily parallelizable and will benefit from the PROCESS computing approach described in Section 3.

**UC#4**: The first step of UC#4 is to generate a standard ancillary sales dataset for airlines. This step involves ingesting data in two manners: (a) using a batch load and (b) using streaming (data). Both data collection tasks are independent of each other (i.e. we plan to use separate data stores, like HDFS etc.). UC#4 is quite different from other PROCESS use cases from a data access point of view, it will serve to validate the idea of application centric data programmable infrastructure described in Section 2. In the upcoming period, the pilot use case will be extended with a model for ancillary pricing connected to each of the data stores, which needs to be designed in more detail. At this moment, we are considering a neural network approach (see also D4.1, chapter 1.4.6, figure 10). Therefore UC#4 will have similar requirements as UC#1 and will benefit from all the work that has been spent in porting UC#1.

**UC#5**: The pre-processing part of UC#5 mainly consists of mathematical operations, in detail error-calculations on millions of measurements values. This calculation is embarrassing parallel and therefore will benefit from the proposed micro-infrastructure. UC#5 data access can be easily parallelized, different parts of the datasets can be transferred and processed at different sites in parallel. Also, the metadata handling is extremely simple and can be covered by the proposed solution. For the software in development, which will use colour- and radar-image processing, the main calculations run on separated grid sectors, which makes it also easy to distribute over many computation centres and data storages. This use case will profit from the containerized pipelines; also, its software will be containerized into Singularity container and so integrate very well into the architecture.

## 2.5 Meta Data

In addition to the regular binary data (BLOBs) which is going to be stored in the Data Service described above each use case includes some portion of numeric/textual data which in addition to storage and retrieval needs to be searchable in an efficient and straightforward way. For example, the UC#1 in addition to medical images for further processing features additional information in form of XML, CSV or TXT files which may be stored in the Metadata component described here. In similar fashion, the UC#2 tape archives contain both measurements (data) as well as accompanying metadata. In case of the UC#3 precise data and metadata structure is going to be specified during the course of the project and will be specific for user groups, however, it's likely that both data and metadata components would be required. For the UC#4, we have already discussed the possibility of storage and usage of at least part of the generated data in metadata storage. Due to the nature of this use case, most data will be numerical and needs to be queryable. Finally, the UC#5 also features the mix of raw imaging data (best suited for the Data Service) as well as textual metadata describing the properties (e.g. of sensors), currently stored in the XML files, which also may be moved to the Metadata Service for computation and further access.

Metadata framework architecture was refined to address new challenges arising in the exascale use cases. Most important factor in this application was allowing the framework for scalability, in order to meet higher demands for throughput and repository load.

The new architecture is based on the concept of decoupling metadata repositories used by different user groups / different use cases in order to ensure the independence of the load they are imposing on the underlying infrastructure. A single repository is a scalable Docker service deployed in a Docker cluster. A repository consists of a cluster of scalable NoSQL database shards and sharded REST interface servers with a load balancer on top. Each repository provides a User Interface component - a browser allowing for simple metadata retrieval / search / insert / update operations. In order to implement presented architecture following technologies were used: MongoDB, RESTheart, HAL browser. The new architecture has been partially implemented in the pilot.

The main challenges to address during implementation:

1. Architecture - integration between a single instance of User Interface being the IEE portal and multiple instances of Repository Browsers. As a user entry point for the service will be integrated with the main User Interface of the execution platform, the method of integrating these components has to be addressed. Each user will be allowed to access several metadata repositories - as each repository is dedicated for a use case. Each repository (thus each repository browser User Interface) is deployed on resources dedicated for the use case, therefore it constitutes a separate service. Browsing different repositories from the perspective of a user entering via a single-entry point (the UI portal) boils down to accessing different web servers deployed in different infrastructures. That is why composing the views provided by different web servers into a single User Interface in order to deliver a uniform user experience is a challenge that needs to be addressed.
2. Security - metadata browser, metadata API and web user portal require an integrated authentication/authorization solution. There are challenges on several levels that arise in this domain. A comprehensive metadata access solution should address issues of permission management on a repository level, user group management (or integration with

an AAI[21] framework providing group management) and procedures and tools for granting access to users, in order to allow different user groups to control the access to metadata resources. Integration with data access framework to reflect the policies implemented on their side is also a viable solution.

3. Docker Swarm address space - a challenge for new resource providers in order to allow DataNet for scaling repositories. DataNet architecture envisions a procedure, allowing to engage a combination of computing and storage resources in order to deploy a metadata repository in an environment provided for example by a use case provider. These resources can be used either as a supplement for the resources provided by the DataNet platform (allowing for further repository scalability) or as a dedicated environment for repository deployment. Incorporating in-house resources in an already established Docker Swarm cluster requires the new resources to be 'visible' as worker nodes by Docker Swarm master node and the master node. This circumstance needs to be addressed as a requirement for the providers of the new resources.
4. Deployment and bootstrap of new repositories - determining requirements on operating system level / docker swarm API permissions that should be granted to the DataNet in order to spawn containers in existing and deploy new repositories. This challenge is connected with the previous one. Once a Docker Swarm cluster discovers the new nodes, that can be used in order to scale out a repository service, all the system level / API permissions required to perform bootstrap/scaling of the service on top of the new nodes need to be granted to the DataNet system. As these aforementioned functionalities will be developed in the final version of the DataNet platform a complete set of requirements for the computing and storage resources providers need to be determined - this is a challenge that needs to be addressed in the target implementation.

In the pilot version of the software, we focused on the reimplementation of the central part of DataNet architecture - the metadata repository. The approach we adopted will allow for further development in order to meet new requirements in the final version. Metadata repository comprises of a storage component being a NoSQL database and a RESTful programmatic interface. Both are deployed as a separate Docker container. The implemented solution is fully functional metadata repository, although it does not allow for scalability at this stage. The next steps of implementation will focus on completing the presumed vision of the architecture

## 2.6 API

Every UC or user will have a custom WebDAV endpoint on *dav://data[xx].process-project.eu:[xxxx]*. The WebDAV integrates to the Interactive Execution Environment (IEE) through a custom authentication method using JSON Web Tokens (JWT). Every WebDAV call is furnished with a signed JWT token from the IEE. Our custom WebDAV is able to verify the token signature using accepted public key. The payload carries the user email and name and therefore our WebDAV can verify users against certain WebDAV points.

Furthermore, the lifecycle of a micro-infrastructure is managed through a set of REST APIs which are currently being developed. The API endpoints manage the user credentials to remote data storages, the creation of a custom micro-infrastructure so as to attach additional storages, retrieving user endpoint information, and exposing advanced programmability

[21] AAI: Authentication and Authorization Infrastructure

features such as programming custom logic pods for the data micro-infrastructure. Authentication to the REST APIs will be through JWTs

# 3 Computing services

## 3.1 Architecture

The overview of the Architecture of the Compute subsystem is shown in Figure 10.

The presented diagram is founded upon the assumption that applications which require deployment on extreme-scale computational resources may come from a variety of sources, including in particular those enumerated in the previous part of this section, and that an exascale computing environment is unlikely to emerge in the form of a single computing centre operating a monolithic HPC infrastructure (i.e. a cluster or cloud site). Consequently, we foresee the need to distribute computations across multiple (possibly heterogeneous) sites. Fortunately, however, the consortium is equipped with the technologies and expertise to tackle both these issues. At the same time, our goal is to provide a unified entry point to the system capable of running computations on the various types of the infrastructures such as HPC (both directly as well as in the containerized form on Singularity nodes) or Clouds (public or private).

In the first prototype, we're going to provide the above system fully integrated and capable of running computations on the single HPC system (at Cyfronet). The ability to utilize multiple clusters (and other infrastructures) would follow in the future iteration.

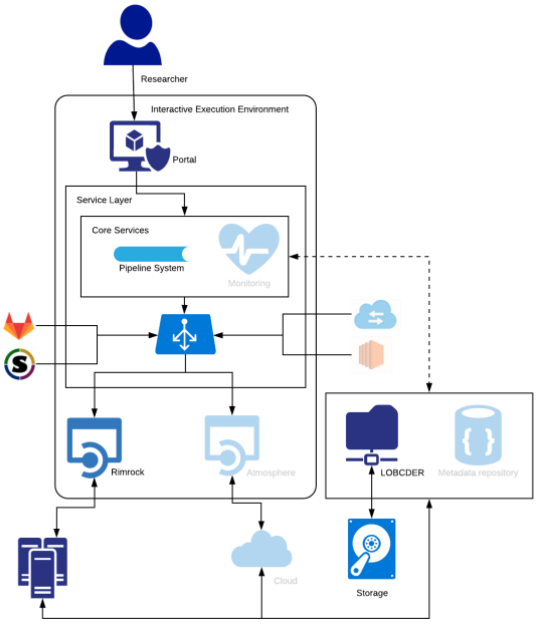



*Figure 10: Architecture of the PROCESS Computing Services*

The Interactive Execution Environment is intended as a subsystem through which PROCESS computations can be launched and their results monitored. In accordance with the Description of Work (DoW), the environment is based on the "focus on services and forget about infrastructures" idea. The underpinnings of the execution environment are provided by a number of existing systems, of which the most notable ones are the Collage infrastructure and the EurValve Model Execution environment, capable of executing HPC computations on distributed resources for the EurValve VPH collaboration.

The following aspects should be noted in the context of designing an execution environment for exascale applications:

- In terms of High-Performance Computing, current large-scale infrastructures tend to be dominated by computing clusters located at large data- and computing centres, some of which are involved in the PROCESS collaboration.
- As remarked in D5.1, a simple calculation, which aggregates the computation power of the 2017 Top500 supercomputers, can only reach 418 Petaflops. Thus, it is not expected that a single computing cluster will be able to provide 1 exaflop of computing power, and accordingly, an exascale infrastructure must be able to harness the power of multiple clusters spread across geographically distributed computing centres.
- A notable offshoot of classical HPC, made possible by the increasing availability of virtualization solutions, is cloud computing. While not traditionally thought of as high-performance resources, cloud computing has found an increasing number of applications in the so-called “long-tail science” (or “midlevel science”), and technologies have emerged which enable computations to be farmed out to a large number of cooperating cloud-based virtual machines in an HPC-like fashion.
- There is increasing usage, in scientific computing, of container infrastructures, and the popularity of technologies such as Docker and Singularity is on the increase. Containers provide an additional layer of virtualization, facilitating deployment of self-contained computational solutions on arbitrary supporting resources, including HPC (as will be discussed later on in this section).

In summary, the basic assumptions which drive the design and implementation of the PROCESS Interactive Execution Environment are as follows:

- **Interactive**: The environment must provide a set of user-friendly interfaces, including GUIs (but also APIs for programmatic access), as already explained in 71.1 – page 8.
- **Execution**: The primary aim of the IEE is the execution of applications in a massive-scale computing environment. Given the great heterogeneity of PROCESS use cases, it is necessary for the environment to remain as generic as possible, and not bound to the requirements and specifics of any particular application. In addition, the environment must provide features to monitor the status of execution and communicate results back to the user.
- **Environment**: The IEE must manage diverse computational resources and be indefinitely extensible in the sense that additional computational infrastructures must be easy to integrate with the existing system, and the system must be able to easily deploy computational tasks to any arbitrary infrastructure to which it has access. The latter requirement is necessary for the infrastructure to achieve exascale-level functionality.

When designing components of the IEE care must be devoted to eliminating any potential bottlenecks which would prevent the environment from scaling out. This concerns both the number of computations present in the system, as well as the number of computational resources available to the system.

In line with the Project’s DoW, the platform is also expected to offer a set of web-based user interfaces through which its features should be conveniently accessible. In addition, the PROCESS infrastructure is expected to be accessed programmatically, and a corresponding set of APIs needs to be defined and established.

In designing the PROCESS UIs, a set of assumptions has to be made with regard to how the computing resources can be integrated to ensure extreme scalability, and how, in turn, these resources are going to be accessed so as to preserve – inasmuch as possible – the design and structure of existing applications (i.e. without having to reimplement the use cases from scratch).

## 3.2 Implementation

In typical workflows, the researcher would access the portal of the Interactive Execution Environment (IEE) which would allow straightforward preparation of the computation. This includes the ability to choose appropriate execution scripts from the repository and provide inputs. Such preconfigured pipeline may be run on the infrastructure via Rimrock component. The orchestration would be done with Cloudify described more in detail in the following Section.

This IEE would obviously interact with other platform components such as the WebDAV file store - both directly in the portal's file browser as well as indirectly from the computation code itself via the API.

The details of the implementation are shown as a demonstration in the separate Deliverable D6.1.

## 3.3 Evaluation and limitations

### 3.3.1 Heterogeneity of the infrastructures

One of the most important limitations of the exascale system especially based on multiple sites is its heterogeneity.

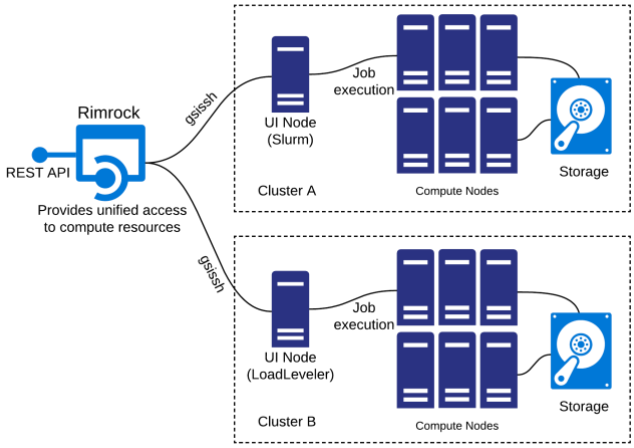



*Figure 11: Use of Rimrock to enable access to heterogeneous HPC resources*

Each cluster may use a different mechanism for UI Node access (such as GSI-SSH, regular SSH, Grid Middleware etc.), a different queuing system (like SLURM, LoadLeveler, PBS). To provide the best interoperability between different clusters from the beginning we will use the component called Rimrock as shown in Figure 11.

As shown in the figure it provides unified access via the REST API which may be triggered by components such as Cloudify. It also features multiple pluggable backends which may be customized to fit multiple sites needs as required.

### 3.3.2 Software dependencies

Infrastructure heterogeneity is not only a big challenge when attempting to run computation in big shared environments such as the HPC. Each use case requires a unique set of the software packages with their own dependencies (such as libraries in a proper version). For shared infrastructures such as typical HPC clusters, this software needs to be either preinstalled by the system administrators or build from source for each use case. The former is not very flexible and may cause conflicts between software packages and the latter requires significant work to enable each use case on each available Cluster. One of the possible solutions is the use of Cloud infrastructures (which we also plan in the subsequent phases of the project) providing a dedicated environment (instance) for each tenant. However, in the scope of HPC, especially exascale grade HPC technical overheads caused by the typical full virtualization used in the Clouds, even if not very significant, may still be a problem. As a solution, we propose wrapping the software into containers which then may be run with nearly no overhead (as shown earlier in this document) on the HPC infrastructures. Due to the shared nature of the HPC system and its security requirement usage of some commonly used container technologies such as Docker is usually not possible (we still plan to use Docker in the Cloud), however other technologies such as Singularity are well supported.

## 3.4 Applicability to use cases

The IEE as an entry point for the platform enabling users to prepare (e.g. provide inputs) configure and run computational pipelines would be applicable to all use cases with UC#1 and UC#2 being most promising candidates for the early adopters. To allow reaching this goal we decided to focus on platform providing a straightforward mechanism to prepare (inputs selection and parameters configuration) and run computations as a Singularity container. Use of the containers is not required for HPC, however, it enables the UC owners to provide a ready-made package with all dependencies intact. Without containers, we would need to coordinate with the HPC Centres actions to ensure all dependencies are pre-installed. We would also need to put considerable effort to adapt the UC provided code to be appropriate for each HPC Cluster. Even on the scale currently available for the projects (multiple clusters provided by four partners), it would be a time-consuming task and for foreseeable future, on the road to exascale, it would quickly become infeasible. The choice of Singularity containers over other technologies such as Docker for the classical HPC part has been prompted by the fact that it is better suited for the shared environments in general (due to different security/privilege paradigm) and also specifically HPC systems (due to better support for hardware common in such environments). Obviously above benefits also influenced the decision of the HPC Centres administrators to deploy Singularity. Regardless of that fact, we are still planning to enable usage of Docker containers (where Singularity is not the best option) in the future, but only for non-HPC tasks (such as some data pre-processing) running on Clouds.

As already mentioned at the beginning we were focusing on the UC#1 and UC#2. The UC#1 seems to be a good candidate to be deployed as a container, as it involves complex requirements and is planned to run on advanced infrastructure involving multi-node GPU cluster. On the other hand, a different way of handling the GPU resources in Docker (which was originally used for this use case) and Singularity (which is required for our platform as

described above) forced us to undertake some work required to adapt it. There is still some work ahead to ensure proper operations of the code in scalable fashion on multiple Singularity containers running as jobs on the HPC Cluster, however, we have finished initial work paving the way to meet that goal. At the same time, we have started work on deploying the UC#2 code on the HPC using the Singularity container provided by the use case code developers. Additionally, we have started discussing with the UC#4 representatives' aspect related to the deployment of this use case code in form of the container. As a first step, we began the work on the deployment of the first component of the UC#4 (data generator) using the container provided by the use case developers.

In addition to the work described above, we have prepared a workflow of tasks needed to prepare the use case code as Singularity container applicable to all use cases. This plan has been discussed during the meeting with all partners and will be applied for further development. Deviation from the plan is expected only when it is absolutely necessary (such as when due to licensing or other legal reasons part of the use case code must reside in specific form e.g. preinstalled on single HPC Cluster). Nevertheless, even in this case, we are going to try to containerize the open components of the software and facilitate data exchange between it and the closed part.

### 3.5 API

The IEE component main function is to serve as an entry point to the PROCESS platform and as such its main interface is a Graphical User Interface (GUI) in a form of a Web Portal. However, in addition to that IEE is also offering REST-based API for management functionality (including its security platform). Also, internal components of the platform such as Rimrock or Atmosphere offer their own REST-based APIs.

# 4 Service Orchestration and User Interfaces

## 4.1 Architecture

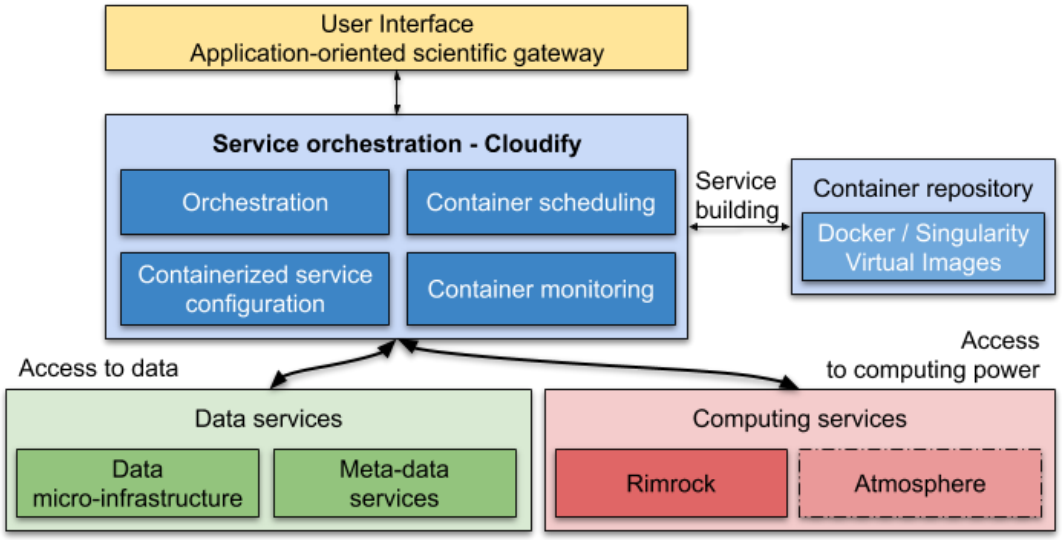



*Figure 12: PROCESS Service Orchestration*

Following the container centric adopted in PROCESS, the User Interface is designed as containerized microservices. Containers provide lightweight and efficient virtualization of computing resources, easy to configure and deploy while being isolated from the underlying

infrastructure. The flexibility of this approach is that either several containers can be used to constitute a single tool or several microservices can be contained inside a single container. From the architecture viewpoint, the PROCESS User Interface is built based on four principles:

- **Modularity** means that individual functional components will be provided and composed as mutually compatible, reusable and extensible microservices.
- **Reusable** means that maximizing reuse capacity and functionality of existing components from PROCESS subsystems such as data management, computing management, and service orchestration as well as auxiliary modules.
- **Easy-to-use** means user-driven design strongly based on end-user's application requirements. The aim is to provide comfortable presentations of PROCESS e-infrastructure to end-users.
- **Openness and interoperability** mean that whenever possible, to use and extend assessed Open Source platforms, thus possibly limiting additional costs of licensing.

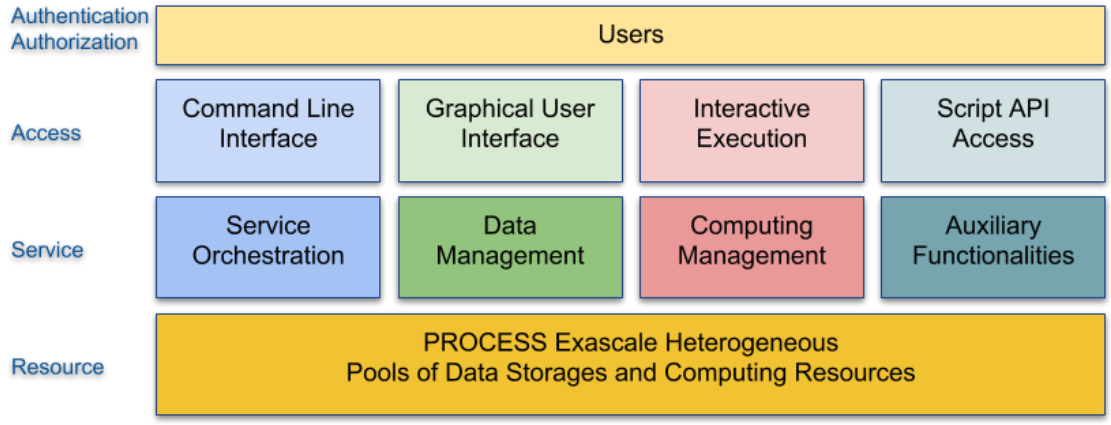



*Figure 13: PROCESS User Interfaces*

PROCESS architecture provides users with four ways (Figure 13) of access and utilization of designed e-infrastructure exascale services and sources as follows:

- **Command-Line Interface (CLI)** is the primer user-to-system interaction in every system. It is a standard and secure way to configure machines, perform tasks efficiently, easy access to applications and their features as well as flexible data management.
- **Graphical User Interface (GUI)** provides to users a comfortable way to interact with the system through direct manipulation of the graphical elements.
- **Interactive Execution Environment (IEE)** is a special element to execute and support application instance executions in PROCESS exascale heterogeneous environment.
- **Script API** and auxiliary functionalities.

Authentication and authorisation of the user are performed by a dedicated service.

## 4.2 Implementation

The service orchestration is composed of the following components:

- **The TOSCA templates**: describe a topology of cloud-based services, their components, the relationships, and the processes managing the services. The topology is a graph of template-nodes modelling the components and template-relationships modelling the

relations between these components. Both the nodes and relationships types define operations to implement the services lifecycle; the orchestration engine can invoke these operations when instantiating a service template. Topology templates are specified in YAML and permanently stored on the orchestration service.

- The input parameters of services: they describe concrete deployment of a specific service (e.g. software versions, service dimensions, service credential, target cloud). Input parameters are specified via GUI or in a file as parameters of CLI during the service deployment.
- The artefacts: they contain the implementations of lifecycle operations of the components in TOSCA templates. Artefacts are usually implemented as scripts or Ansible roles. Artefacts are stored on the orchestration service or public repositories (e.g. GitHub).
- The Cloudify orchestration engine that will process TOSCA templates uses the mentioned lifecycle operations to instantiate single components at runtime, and it uses the relationship between components to derive the order of component instantiation.

At the moment, the PROCESS User Interface is partially realized as subsystem components in the initial state. These four ways to accesses/utilise the exascale e-infrastructure (CLI, GUI, Interactive and Script API) ensure that users will get full support and services from underlying PROCESS subsystems. The user-driven realization details established on the concrete user requirements about UI will be realized in the incoming Deliverable 4.3 as a part of updated requirements analysis and validation. The UI modular and reusable approaches, in the current states, are as follows:

- **Data Management** (Section2.2-page 11, Implementation of Data Management) considers exascale data architecture with two separate data management stacks: one dedicated to user access and the second dedicated to application access. The data architecture is exascale-oriented, scalable and distributed over multiple sites.
- **Computing Management** (Section 3.1-page 23, Architecture and Section 3.2-page 25 Implementation of Computing Management) is going to cover the heterogeneity of underlying high-performance infrastructures through REST API portal of the IEE. The portal provides accesses as CLI, Web UI and Script API with typical scientific workflows for application computation preparation included references to input data in repositories.
- **Service Orchestration** (Section 4.5-page 31, API for Service Orchestration) is built for pluggability and evolvability, makes it possible to manage and automate services across environment while maintaining governance and control. The management is currently realized through Cloudify CLI and GUI (CM).
- **Auxiliary Functionalities** e.g. monitoring and visualisation microservices, interactive programming microservices, predictive microservices, etc. The UI realization of auxiliary functionalities will be integrated with the functionalities of the Computing Management (Section 3-page 23) in time.

## 4.3 Evaluation and limitations

TOSCA templates are a standardized and efficient way to describe topologies of services and their life-cycles. In combination with Ansible roles they can be used to deploy complex services like Spark, Kubernetes or HPC clusters in clouds.

Cloudify can use the TOSCA templates to manage whole life-cycles of services including initialization, deployment, monitoring, auto healing and un-deployment. As Cloudify is open-sourced and extensible via plugins, it can be adapted to new computing environments.

TOSCA templates are not designed for executing single computation from end-users. It is intended for deployment of services like Jupyter portal or HPC clusters that users can later submit their computation to the services. The users can make a request what computation environment they need for their applications and the service orchestration will deploy the corresponding service to cloud and give access to them to users.

## 4.4 Applicability to use cases

For use cases in PROCESS, the types of services to be deployed by Cloudify are the following:

a) Environments for application development and testing: Cloudify can be used for deploying the complete environments for application development, testing, performance tuning, and verification. One of the reasons for deploying application development environments in the Cloud is that it is instantly accessible, replicable, repeatable and shareable, without complex library installation and setting. One of the examples of the development environment is Jupyter notebook that is used by UC#1. Applications in such environment are easily shared between different developers, delivered to end-users or make a demonstration for public audience

   Another reason for deploying application environments via Cloud is access to specialized hardware like accelerators, high memory or multicores, that are not always available locally. For examples, GPU accelerators are requested by UC#1 and UC#2, multicores computation are utilized by UC#3 and UC#5.

   Some of the use cases, including UC#1, UC#2, UC#3 use a large amount of RAM. If there is not enough memory on the target execution environment (e.g. on a given HPC cluster), the application execution may fail or has severe performance degradation due to memory swapping. Deploying an application environment in Cloud with similar hardware configuration as the target execution environment will allow developers to adjust application settings and make performance tuning for obtaining optimal performance on the target environment.

b) Virtual execution environment: Cloudify can deploy a complete virtual execution environment like virtual HPC or Spark clusters for use cases. For examples, the Apache Spark is requested by UC#4 and Kubernetes clusters are also used by data services. Such an environment can be deployed as virtual infrastructure and offered as services for use cases.

   Cloud-based execution environments can provide additional computational resources besides HPC platforms for use cases. Using Cloud-based virtual execution environment allow instant access to resources which is suitable for UC#3 where the impacts of the risks must be evaluated as soon as possible.

c) Dynamic deployment of data pre-processing services: The data pre-processing services from WP5 will be deployed not only statically, inside the distributed computing infrastructure of PROCESS. One of the goals of WP7 is to enable a dynamic deployment of pre-processing workflows near the data. This will allow applications to pre-process data in place, without having to move complete large datasets first. The dynamic deployment can occur in pre-prepared, safe containers, trusted by the data owners, which will shield

the data from any potential damage or misuse by the deployed data services. For this, the DISPEL Gate technology is useful, as it allows to prepare an environment in which custom services can be deployed, and data processes can be executed, while the data are accessed only through known and safe services acting as connectors.

### 4.5 API

Cloudify has a REST API for all operations related to service orchestration. This REST API can be divided into several sections, the most important ones are the following:

- Blueprint: management of TOSCA templates, e.g. upload, download, list, delete.
- Deployment: deployment of services and management of already deployed services.
- Execution: executing workflows defined in TOSCA templates on concrete deployment, e.g. install, restart, uninstall

The details of REST API is described at https://docs.cloudify.co/4.5.0/developer/apis/rest-service/

The API will be used by command-line clients (CLI) or scientific gateways (GUI) for deployment and management of services. The services need to be described in TOSCA templates (blueprints) and uploaded to the Cloudify server before deployment using blueprint API. After that, the users can deploy/undeploy instances of services described in the blueprints via deployment API or execute a specific workflow, e.g. restart the service running in the cloud.

# 5 Dissemination

To disseminate the results of PROCESS project, we have decided to use the Research Software Directory[22]. The Research Software Directory promotes the exchange and the re-use of research software across project and eScience scientific domain (Figure 14). Research Software Directory is populated by software developed in the context of more than a hundred projects in eScience which can be characterised as both CPU and data intensive applications. Research Software Directory constitutes a potential communication channel to promote and reach out potential scientific communities

[22] https://www.research-software.nl

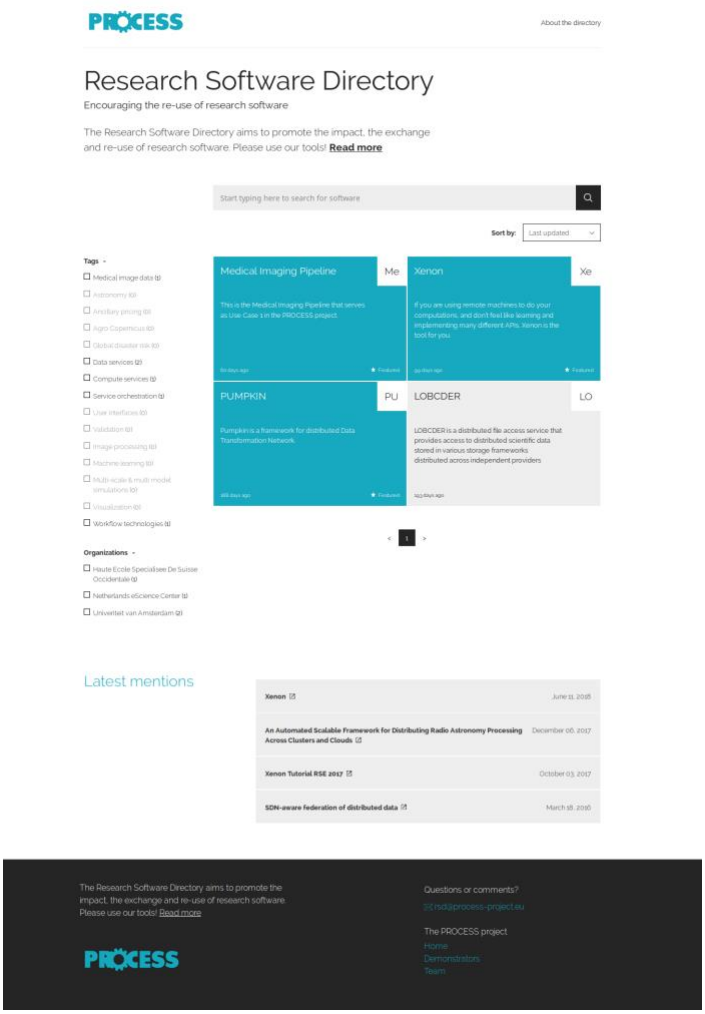



*Figure 14: PROCESS software directory*

To finish this report, we include the list of international events, in which the PROCESS work has been presented:

1. Bobák, Martin - Belloum, Adam S. Z. - Nowakowski, Piotr - Meizner, Jan - Bubak, Marian - Heikkurinen, Matti - Habala, Ondrej - Hluchý, Ladislav. Exascale computing and data architectures for brownfield applications. In 14th IEEE International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD 2018), pp. 461-468, ISBN 978-1-5386-8097-1. Huangshan, China, July 2018.
2. Mikolaj Branowski and Adam S.Z. Belloum. Cookery: A framework for creating data processing pipeline using online services. The 14th IEEE International Conference on e-Science (eScience 2018), Amsterdam, Netherlands, October 2018.
3. Rosco Kalis and Adam S.Z. Belloum. Validating data integrity using blockchain technology, International Workshop on Resource brokering with blockchain (RBchain), in conjunction with the 10th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2018), December 2018, Nicosia, Cyprus.
4. Krammer, Peter - Kvassay, Marcel - Hluchý, Ladislav. Predicting the probability of exceeding critical system thresholds. CEUR Workshop Proceedings, Volume 2139, 2018, pp. 189-196, ISSN 16130073, (11th International Conference of Programming - UkrPROG 2018), Kyiv; Ukraine; May 2018. Open Access.

5. Krammer, Peter - Kvassay, Marcel - Hluchý, Ladislav: Enhanced data modelling approach with interval estimation. In 16th IEEE International Symposium on Intelligent Systems and Informatics (SISY 2018), Subotica, Serbia, September 13-15, 2018, pp. 179-184, ISBN 978-1-5386-6841-2.
6. Mrnčo, Ivan - Blštak, Peter - Hudec, Peter - Kochan, Matej - Gibala, Tomáš - Habala, Ondrej: Application of advanced information and communication technologies in a local flood warning system. Computing and Informatics (Current Contents), Vol. 37, No. 6, 2018. ISSN: 2585-8807. December 2018. Golden Open Access.
7. Nguyen, Thieu - Tran, Nhuan - Nguyen, Minh - Nguyen, Giang: A resource usage prediction system using functional-link and genetic algorithm neural network for multivariate cloud metrics. The 11th IEEE International Conference on Service-Oriented Computing and Applications (SOCA 2018), Paris, France, November 2018.