Multi-Cloud Deployment and Execution of Earth Observation Services

João Pedro Martins Serras

Thesis to obtain the Master of Science Degree in

Information Systems and Software Engineering

Supervisor(s): Prof. António Manuel Ferreira Rito da Silva
Dr. Nuno Catarino

Examination Committee

Chairperson: Prof. Francisco António Chaves Saraiva de Melo
Supervisor: Prof. António Manuel Ferreira Rito da Silva
Members of the Committee: Prof. Miguel Filipe Leitão Pardal

December 2019

Dedicated to everyone who helped me along these years and supported me to be where i am today.

Acknowledgments

At the end of finishing my master's degree in the Computer Engineering field in Instituto Superior Técnico (IST), I realized these were fantastic years with many exceptional experiences in my life. Even though there were dark times, full of work and failures, they also had a positive important impact in life, making me growth as a person. Through the good and hard moments, I was able to meet amazing people who helped me during these years. Thanks IST.

A special thanks to my grandparents and parents, João and Isilda, for making all this possible, allowing me to be where I am today. To carry on to me ethical and professional values. For providing emotional support and guidance through the years, making me who I am today.

Many thanks to my closest friends with who I shared thousands of good and memories, allowing me to relax and have fun.

I want to thank my supervisors, Professor António Rito Silva, Nuno Catarino and even though not present during this phase of my thesis Nuno Almeida, for providing me the chance to develop this thesis and passing me valuable knowledge for my future career.

At last, I want to thank everyone at Deimos, especially Koushik, who supporting me with everything that I needed and for providing guidance during the development of this thesis.

Resumo

Actualmente, com o progresso da engenharia, os dados espaciais estão a ser colecionados com uma elevada velocidade, volume e diversos formatos através de um grupo de satélites. Para os cientistas estudarem e utilizarem esses dados, primeiro são necessárias algumas transformações que requerem a utilização de ferramentas de Observação da Terra (OT).

No entanto, essas ferramentas podem consumir muitos recursos, restringindo o uso a uma comunidade pequena e altamente qualificada. Com o surgimento da computação em nuvem, os cientistas não precisam de se preocupar com o custo de gerenciamento de hardware. Além disso, a computação na nuvem liberta os cientistas de compreender a infraestrutura dos equipamentos necessários. Como resultado, a computação na nuvem é a solução ideal.

Os procedimentos de exportação das ferramentas de OT para ambientes de Cloud Computing surgem com alguns desafios críticos. É necessário empacotar essas ferramentas, criar assim como configurar uma instância na nuvem e expô-las ao contexto da web. Deste modo, o nosso objectivo é automatizar cada uma dessas etapas, explorando as tecnologias já existentes.

O nosso foco inicial é a criação de um pipeline de entrega contínua de alto grau. Este será o principal especto funcional desta tese. Além disso, o custo monetário ainda será levado em consideração. Isto é, mantê-lo a um nível mínimo. Com isso em mente, pretendemos utilizar um trabalho já existente, intitulado de Mecanismo de Fluxo de Trabalho para Serviços de Observação da Terra para ajudar na validação dos argumentos e dados resultantes da execução das ferramentas de OT, ao implantar estas na nuvem, evitando erros desnecessários durante a execução.

Palavras-chave: Deployment na Nuvem, Provisionamento, Empacotamento de Ferramentas de Observação da Terra, Observação da Terra, Serviços de Processamento de Rede, Mecanismo de Fluxo de Trabalho

Abstract

At this day and age, with the progress in engineering, spatial data is being harvested at a high speed and volume with several formats from a continuously collection of satellites. For scientists to study and utilize this data it first requires some transformations from Earth Observation (EO) tools.

However, these tools can be resource-heavy, restricting the use to a small and highly skilled community. With the arise of Cloud Computing, scientists do not need to worry with the cost of managing hardware. Furthermore, Cloud Computing releases the scientist from the burden of understanding the infrastructure of their equipment. As a result, Cloud Computing became the natural solution.

The procedures of deploying the EO tools on Cloud Computing environments emerges with some critical challenges. It requires to package these tools, provision as well as configure a cloud instance, deploy and expose them to the web context. Therefore, we aim to automate each of these steps by exploring the already existing technologies.

Our focus will begin with the creation of a high degree continuous delivery pipeline. This will be the main base functional aspect of this thesis. In addition, the monetary cost will still be taken into consideration. In other words, how to maintain it to a minimum. With this in mind, we aim to utilize an already existing work, titled Workflow Engine for Earth Observation Services to help with the validation of inputs/outputs when deploying the EO tools to the cloud, avoiding unnecessary runtime errors.

Keywords: Cloud Deployment, Provisioning, Earth Observation Tool Packaging, Earth Observation, Web Processing Service, Workflow

Contents

Acknowledgments v

Resumo vii

Abstract ix

List of Tables xv

List of Figures xvii

Acronyms xix

1 Introduction **1**

1.1 Problem Definition 2

1.2 DevOps 3

1.3 Solution 4

1.4 Validation 5

1.5 Contributions 6

1.6 Thesis Outline 6

2 Background **9**

2.1 EO Tool Packaging 9

2.1.1 Package Management Systems 9

2.1.1.1 Red Hat Package Manager 10

2.1.1.2 Advanced Packaging Tools 10

2.1.2 Containerization 11

2.1.2.1 Linux Containers 11

2.1.2.2 Docker 11

2.2 Cloud Deployment 12

2.2.1 Configuration and Management Tools 12

2.2.1.1 Chef 13

2.2.1.2 Ansible 13

2.2.1.3 Kubernetes 13

2.2.2 Server Provisioning Tools 14

2.2.2.1 Terraform 14

2.2.2.2 Openstack-Heat 14

2.3 Web Processing Services . . . 15
2.3.1 Web Processing Service Transactional . . . 15
2.3.2 CRIM WPS . . . 16
2.4 Thematic Exploitation Platforms . . . 16
2.5 Workflow Chain . . . 17
2.5.1 Workflow Engine for Earth Observation Services . . . 18
2.5.2 Summary . . . 19

3 Tools Analysis 21
3.1 EO Tool Packaging . . . 21
3.2 Cloud Deployment . . . 23
3.3 Web Processing Services . . . 24

4 Implementation 27
4.1 System . . . 27
4.2 Security . . . 28
4.3 MCDEEOS . . . 29
4.3.1 EO Tool Packaging . . . 29
4.3.1.1 EO Tool Wrapping . . . 29
4.3.1.2 Web Processing Service Encapsulation . . . 33
4.3.1.3 Retrieval of Web Processing Services XML Description . . . 34
4.3.1.4 Web Processing Services Image Storing . . . 34
4.3.2 Workflow Execution in Cloud . . . 34
4.3.2.1 Cloud Provisioning . . . 34
4.3.2.2 Cloud Configuration . . . 37
4.3.2.3 Augmenting the system with more cloud providers . . . 40
4.3.2.4 Disk Size Modes . . . 40
4.3.2.5 Workflow Orchestration . . . 41
4.3.2.6 Augmenting the system with more workflow engines . . . 42
4.3.2.7 Execution Modes . . . 44
4.4 MCDEEOS-WEB . . . 45
4.4.1 EO tool Packaging . . . 45
4.4.2 List WPSs . . . 48
4.4.3 Workflow Execution . . . 49
4.4.4 Register WPSs . . . 51
4.4.5 Remove WPSs . . . 52
4.4.6 Discover WPSs . . . 52
4.4.7 List WPSs from Catalogs . . . 53

5 Validation **55**
5.1 Goals . . . 55
5.2 EO tool Packaging . . . 55
5.2.1 Manual Steps . . . 56
5.2.2 System Steps . . . 58
5.2.3 Errors . . . 58
5.2.4 Execution . . . 59
5.2.5 Validation Script . . . 60
5.2.6 Results . . . 60
5.2.7 Analysis . . . 61
5.3 Workflow Execution . . . 62
5.3.1 Manual Steps . . . 62
5.3.2 System Steps . . . 64
5.3.3 Errors . . . 64
5.3.4 Execution . . . 65
5.3.5 Validation Script . . . 66
5.3.6 Results . . . 67
5.3.7 Analysis . . . 67

6 Conclusions **69**
6.1 Achievements . . . 69
6.2 Limitations . . . 70
6.3 Future Work . . . 70

Bibliography **73**

A XML Describe Process **75**

B Documentation **77**
B.1 MCDEEOS Guide . . . 77
B.1.1 EO Tool Packaging . . . 77
B.1.2 Workflow Execution . . . 84

C OGC Technical Committee Meetings **89**
C.1 24 January 2019 . . . 89
C.2 27 November 2019 . . . 92

List of Tables

3.1 Characteristics of Application Packaging Tools 22
3.2 Characteristics of Cloud Provisioning Tools 24

4.1 Deployment time using self made image in seconds 39
4.2 Deployment time using base ubuntu image in seconds 39

5.1 EO Tool Packaging times recorded from the system validation 61
5.2 EO Tool Packaging errors recorded from the validation 61
5.3 Workflow Execution times recorded from the system validation 67

List of Figures

1.1 Continuous delivery pipeline from [4] 4

2.1 Overview of a Workflow Pattern 17
2.2 Identify illegal mining activities workflow 18

4.1 List of WPSs present in a catalog in the system 28
4.2 EO Tool Packaging CD Pipeline 34
4.3 Workflow execution state machine 42
4.4 Camunda graphical user interface 42
4.5 BPMN workflow diagram 43
4.6 Modified BPMN workflow diagram 43
4.7 Information required about the EO tool for its packaging 46
4.8 Information required about the EO tool processes for its packaging 47
4.9 Information required about the ComplexOutputData for the EO Tool Packaging 47
4.10 List of WPSs present in the system 48
4.11 List of the WPSs belonging to a user 48
4.12 Description of a WPS name wps_crop 49
4.13 Dashboard for the workflow creation 49
4.14 WPS process specification 50
4.15 Console displaying the errors detected during the workflow validation 50
4.16 Cloud information required for the workflow execution 51
4.17 Interface for the WPS catalog registration 52
4.18 Interface for the WPS catalog removal 52
4.19 Interface for the WPS catalog discovery 53
4.20 List of WPSs present in a catalog in the system 53

5.1 State machine describing the manual procedure for the EO Tool Packaging 57

B.1 EO Tool Information 78
B.2 Dockerhub Repository 78
B.3 Process Information 79
B.4 Input Data Buttons 79

B.5 Literal Input Data 80
B.6 Complex Input Data 81
B.7 BoundingBox Input Data 82
B.8 Output Data Buttons 83
B.9 Complex Output Data 83
B.10 Workflow Dashboard 84
B.11 Information for creating the WPS in the dashboard 84
B.12 Process Information of the WPS 85
B.13 Console to display validation errors 85
B.14 Cloud information required for the workflow execution 86

Acronyms

ADES Application Deployment and Execution Service.

APT Advanced Packaging Tools.

AWC App Management Client.

CD Continuous Delivery.

CLI Command Line Interface.

CMT Configuration and Management Tool.

CRIM Computer Research Institute of Montreal.

CRS Coordinate Reference System.

DIAS Copernicus Data and Information Access Services.

EO Earth Observation.

GB Gigabytes.

GCP Google Cloud Platform.

GUI Graphical User Interface.

HCL HashiCorp Configuration Language.

Heat Openstack-Heat.

HOT Heat Orchestration Template.

IST Instituto Superior Técnico.

JWT JSON Web Tokens.

LXC Linux Containers.

MCDEEOS Multi-Cloud Deployment and Execution of Earth Observation Service.

OGC The Open Geospatial Consortium.

OS Operating System.

OWS OGC Web Services.

PMS Package Management Systems.

RPM Red Hat Package Manager.

SPT Server Provisioning Tool.

TB Terabytes.

TEP Thematic Exploitation Platforms.

VM Virtual Machine.

VPC Virtual Private Cloud.

WE Workflow Engines.

WEFEOS Workflow Engine for Earth Observation Services.

WPS Web Processing Services.

WPS-T Web Processing Service Transactional.

Chapter 1

Introduction

At the present time, with the increase interest in spatial investigation, the European Union began an Earth Observation (EO) Program entitled Copernicus (former GMES). The purpose of Copernicus is to gather and monitor data gathered from a collection of satellites. This data is then used to produce products through the use of EO tools. As a result, products are then exposed to the public with the intention of improving scientific fields. Notably the data retrieved by the satellites are in the order of petabytes per year, hence the transformation of this data into higher level products is resource intensive. Building and maintaining complex infrastructures are not feasible anymore. It requires a large monetary investment and a deep knowledge about the underlying infrastructures, making cloud computing a much more appealing solution, since it releases the customers from this burden [1].

Cloud Computing is a shared pool of configurable computer system resources and higher-level services that are delivered on demand to external customers over the Internet. It brings high availability and increased flexibility by adjusting to the customers requirements whenever it is necessary, without regard to where the services are hosted or how they are delivered [2]. Cloud Computing empowers the backup, recovery and sharing of information. All documents are stored in the cloud, a shared environment with restricted access.

Therefore, the European Commission has decided to invest in the development of Cloud Computing platforms specialized for the storage and exploitation of the Copernicus data, Copernicus Data and Information Access Services (DIAS). DIAS is composed by five cloud computing platforms known as Creodias [1], Mundi Web Services [2], Onda [3], Sobloo [4], Wekeo [5]. It eases the procedure of accessing to Copernicus data and information from the Copernicus services by providing data and information access alongside computing resources.

[1] https://creodias.eu
[2] https://mundiwebservices.com
[3] https://www.onda-DIAS.eu/cms
[4] https://sobloo.eu
[5] https://www.wekeo.eu

1.1 Problem Definition

To summarize, a massive amount of data is being gathered by the Copernicus that demand transformations in order to be used. This process of converting data to products is resource expensive, making the use of DIAS the natural solution. However, these data transformation tools were not designed with the usage of DIAS in mind. Exporting them to the cloud environment rises some challenges.

Each tool has a set of dependencies that are required to ensure it is the correct execution. Executing these tools on the cloud oblige these dependencies to be available in the environment. This can be achieved through two methods. Either the cloud environment is configured to contain these dependencies, installing directly the dependencies in the cloud environment, or they are bundle together with the tool itself into a package. Thereby, a new problem arises. How do we package the tools or configure the cloud environment, and which one do we choose?

Succeeding this phase, the application needs to be deployed, transferred to the DIAS, and then executed. However, how do we deploy the tools over the different DIAS? This is another problem that requires to be solved.

Whereas the answer for these questions lie in the computer science field, we must have in mind the target audience of these tools are scientists that do not possess enough knowledge in this field. As a result, it is necessary to spend time to learn how to manage cloud computing. However, the learning process can take quite some time and does not mean that the user will be proficient when using cloud technology. This can also be translated into unnecessary user errors. A simple task of running an application will grow into a nuisance. Ultimately, the single concern of the user should be to execute the EO tools with the desired input.

With all these points, we can state that our main goal is concerned with the usability. Therefore, we aimed to ease the accomplishment of the execution of a desired tool by its users. Instead of forcing the users to adapt to the cloud computing, we developed a system that abstracts the user from this procedure. This removes the cloud learning curve, allows its users to efficiently utilize its resources while minimizing the impact of unwanted errors and increasing the overall confidence in using cloud computing as a solution to the lack of resources.

All things considered, with the intention of removing the cloud learning curve, it is important to denote, the data transforming tools need to be exposed to the users in such way that the user does not need to worry about the cloud management. Consequently, the ideal approach is to expose this tool through the web. In other words, we must be able to accommodate these tools to the web context for the users to utilize them. As a result, a new challenge arises.

It is also essential to have in mind that the package and deployment, is done manually every time it is required to use a tool. Considering the utilization of multiple tools several times a day, in the long run, a great deal of time is spent in this procedure rather than the execution itself. Performing these tasks manually is not reasonable. Thus, automation is required to improve the usability.

1.2 DevOps

As stated before, our main concern is usability. Provide a system that enables the users to execute different EO tools over any cloud provider for example the DIAS, without worrying about the problems that come with the deployment of these tools to the cloud. For this reason, we want to automate every step of the solution for the problem mentioned above in section 1.1, Problem Definition.

In the light of this, it is possible to observe that it fits into the DevOps context. The process of developing software and release it to production can take several hours, if not days, and in the end, it can reveal to be unstable, when each step is done manually.

Therefore, DevOps practices aims to shorten the time between committing a change to a system and the change being placed into normal production [3]. This is done through the implementation of an automated Continuous Delivery (CD) pipeline, providing a high degree of automation [4]. This CD pipeline is represented in Figure 1.1. It can be characterized in three stages followed by the testing of each, except the last phase.

The first stage is triggered by the commit of the newly developed code. Once the code is committed, unit tests are performed over the code to make sure every other function is not broken. While in our case we don't develop new features to the same software, we can translate this phase to the adaptation of the EO tools to the web context. Every time a new tool is committed into our system, it will trigger a transformation process to the tool, that will allow the exposure to the web. Upon the passing the tests, it enters the second stage.

The second stage is responsible for the building of the application. The application binary package is created with all its dependencies. Fresh tests must be executed to verify the application package is well done and meet the desired requirements. This stage perfectly matches the application problem challenge. However, as stated above, we have two approaches, either package the application or configure the cloud environment. Good DevOps practices favor application packaging method. While configuring the cloud environment is a solid choice, it bounds the application to that environment. Simply put, if we change to another cloud environment, we would need to configure all over again from scratch. In addition, the method to configure each cloud environment might vary from environment to environment. Leaving us more prone to errors. Therefore, application package is considered the best choice in regard to DevOps. However, in the DevOps context there are still flexible tools that help with both cases.

The third stage is responsible for the deployment of the application to production. In our case, this is translated to the provisioning of a new cloud instance to deploy the EO tool. Consequently, DevOps practices depend in various tools to aid each step. Each of this stage corresponds to a sub problem mention in the section above. With this in mind, this thesis will rely in DevOps practices to produce the bigger picture of register an EO tool and deploy it to the cloud. Hence, DevOps tools will be explored in the following chapters.

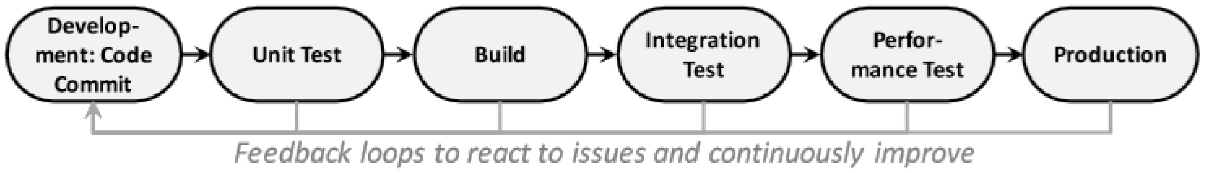


Figure 1.1: Continuous delivery pipeline from [4]

1.3 Solution

Before giving a brief explanation of how we develop our solution, it is important to mention that there were no previous solutions similar to the one proposed, in the EO field, that solved the usability problem. Given an explanation on the DevOps practice, we were able to provide automation regarding the problems specified in section 1.1, Problem definition. However, since our goal is not related to the exposure of a system immediately to production, we developed two pipelines with additional stages.

Our first pipeline focuses on solving the exposure of the data transforming tools to the web context. To solve this challenge, we observed that The Open Geospatial Consortium (OGC)[6] developed a standard interface, Web Processing Service (WPS), that provides the rules on how the input and outputs must be specified as well as the methods for execution and processes descriptions. Therefore, with the help of an open source WPS implementation, we were able to wrap every EO tool into a WPS and execute over the web. This constitutes the first phase of our pipeline. With the challenge of exporting the EO tools to the web context solved, we can move forward in the pipeline to the next phase, package the WPS. In this phase, we encapsulate all the dependencies and applications together in a unique isolated environment that is not dependent on any hardware. Therefore, we can launch the WPS in every cloud environment and execute it without worry about adapting either the WPS or the environment. Minimizing unwanted errors when executing the EO Tool. As we mentioned before, we do not want to instantly release the newly WPS to production. We provide the users with the options to utilize it when they feel its necessary. As a result, the last phase is characterized by pushing the EO tool into a repository. Additionally, this allows for the reusability of the EO tool without needing to re-wrap the tool whenever the user desires to execute it. With the establishment of this pipeline, we are able to remove the constrain of learning how to access cloud environments and manage the dependencies to execute the EO tools. Consequently, the knowledge required from the users to use the cloud decrease.

In brief, the first pipeline solves the adaptation of the EO tools to the web context and the application packaging. Thus, the second pipeline addresses the remaining problem, deploying the application. In order to use the cloud in as an efficient solution, we opted to utilize resource on demand. Simply put, we provision new instances and deploy the application only when the user decides to use it. Hence, the first phase is defined by the creation of the newly instance, with a specific hardware configuration tailored to the EO tools needs. Having the right configuration allows its users to avoid errors that will translate into unnecessary costs due to not having enough resources or having too many resources used. As a result,

[6]http://www.opengeospatial.org

we are able to efficiently use the cloud for the users needs. Once the instance is set up, we require to configure it. Lastly, we need to pull the application from the repository to the cloud environment.

Having in mind we desire to be the most efficient when using the cloud, the users might desire to chain applications. In other words, redirect the output from one application to another application as an input. Doing this process manually is time wasting. In the event of an application finishing its execution, the users are not notified. Therefore, resources are wasted waiting for the users to check the execution result. To solve this problem, we decided to use a workflow engine. The workflow engine will allow the user to specify the chain of applications to our system. Thus, we are able to orchestrate its execution, triggering the second pipeline for the next application when the previous execution finished and redirecting the output right away.

It is important to bear in mind when performing both pipelines extra information is required to ensure the proper execution of both. In the first pipeline, to perform the wrap of the application into a WPS, the users must specify each type of input, output, the command line and the output folder. On the second pipeline, the users must indicate, the workflow, the workflow engine, to which cloud and region each WPS will be deployed and how much space the cloud environment requires. As been noted this is critical information that we can not guess. Therefore, it is required for the users to specify it.

All things considered, the methodology of the pipelines based on the DevOps context while using a workflow engine allows us to provide a system that adapts the cloud technology to the needs of the data transforming tools users.

1.4 Validation

Ultimately, we were able to solve the challenges that arose with executing EO tools in the cloud environment. However, we still need to verify if the system presents superior results than performing this process manually, considering the nonexistence of such system in the EO field.

It is important to bear in mind when developing the system to solve the problems defined above, we desired to reduce as much as possible the cloud learning curve while providing the most efficient service and diminish unwanted errors that converge in a waste of resources, such as time and money. Thereby, to compare our solution with the process done manually, we translate these into qualitative goals. Ease to learn the system, ease to use the system and error tolerance. As a result, we use the efficiency and effectiveness as metrics, as to measure these we will resort to number of errors, time and cost.

Effectiveness is the quality with how the tasks are achieved [5]. This is measured by the percentage of errors amended in our system. To accomplish this, we defined a list of errors, which are the most frequent, and aimed to prevent them from happening. Helping us to assess the error tolerance. As an example, users tend to forget to specify dependencies when packaging or configure the security group when using the cloud. Under these circumstances the application is not able to execute properly and the cloud environment does not have access to the web to pull the application from the repository, nor we have access to the application running in the cloud to execute it. Since it is not possible to predict every error we might encounter, we intend to reduce the amount of errors in comparison to performing the tasks manually.

Efficiency, on the other hand, is measured by the resources needed and wasted to accomplish the task [5]. Notably, we relate resources with the time and costs in the context of this thesis. Using time helps us evaluate how easy it is to learn and use the system, by measuring how much time is spent in learning and using each feature of the system. To be more specific, with regards to evaluate the time of learning to use the system, we will measure the time spent on reading documentations that aid in using the system for each task.

All things considered, we measure these metrics for each step of both pipelines in order to perform a deeper analysis. Regarding the first pipeline we measure how long it takes to perform and learn the wrapping, packaging and storing of the EO tool as well as how many errors are made when performed manually. Following this, we measure time spent learning how to use our system and the time spent in the execution of each step by the system. The same procedure is used for the second pipeline. In other words, we will measure the errors made, the time spent in learning and provisioning the instance for the deployment of each EO tool in the workflow for both cases.

With the results recorded and displayed through tables, we proceed to analyze them. We obtain the average of the time spent per user in both learning and executing the tasks and compare both approaches, the manual and automated. At the same time, we justify why each approach obtain the indicated average time.

1.5 Contributions

The Open Geospatial Consortium (OGC) is an international consortium comprised by several organizations with the objective of sharing, discover and reuse information as well as services related to the diverse EO fields. Every year, the OGC publish Engineering reports. These reports contain the activities that greatly contribute to the EO community.

As already mentioned, our solution comprises in two separate features, the application packaging and the workflow execution using cloud computing. Although both display an impact in the EO community, the execution of workflows using cloud environments was the one with the biggest impact, fitting every category presented by the OGC to participate in the Engineering reports.

As a result, our solution has been tested and verified and demonstrated in OGC Innovation Program (Testbed-15). The technical approach and the summary of the solution is included in the OGC testbed-15 Engineering report for the entire EO community to benefit from.

In the end, the value of our system is clearly identified and will be the baseline for future work in this area.

1.6 Thesis Outline

To accomplish our solution of providing an automated pipeline for both EO Tool Packaging and Cloud Deployment, we will begin by exploring the diverse existent tools for each step of the pipelines in chapter

2. Subsequently, we analyze each tool and compare their strengths and weaknesses in chapter 3. Once, we have selected the finest tools, we begging to describe how we utilized them to support us during the automation process in both pipelines. At the same time, we present the challenges that arose during the development of both EO Tools Packaging and Workflow Execution using the cloud environment in chapter 4. Following the presentation of our solution, we proceed to validate it through usability tests in chapter 5. Ultimately in chapter 6, we present the conclusion of the thesis as well as our greatest achievements, limitations and the future work to be developed.

Chapter 2

Background

Before starting to develop our system, we began by searching the different existing technologies and tools, to help us achieve our goal of providing automation in both EO Tool Packaging, Cloud Deployment and Workflow Execution.

We will start by exploring the different tools to package an application with all its dependencies. Subsequently, we cover the tools capable of provisioning a cloud instance and export the different packed EO tools to the cloud environment. Although, this may seem enough, the users still do not have access to the packed EO tool now running in the cloud. As a result, it is required to exposed them to the web context. For this, we look at the already existing tools developed in the EO community.

2.1 EO Tool Packaging

EO tools are simply applications that implement algorithms to manipulate data and are produced by third-party developers. With the desire to launch and execute them into cloud infrastructures, a major issue came to life. The cloud environment needs to be configured to run the applications, otherwise the execution will fail. Each application has its own requirements that need to be satisfied. As mentioned in [6] [7], these requirements can range from specify the location of information, work areas of the disk that will be used to process configuration files, install libraries and other applications that the main application might require. With this in mind, how do we provide all necessary information alongside the application to configure the cloud environment? The natural answer can be found on the Build stage of the CD pipeline, presented in section 1.2. It is Application Packages. An Application Package encapsulates applications with all its related information to enable its easy management. The process of encapsulation, creating the application package, is done primarily by Package Management Systems or Container.

2.1.1 Package Management Systems

Package Management Systems (PMS) came forth with the urge to automate the process of setting up the environment properly to run the applications. With updates emerging, the number of files increased drastically. Configuring manually the environment, copying all files to the proper place, making sure they have the desired format and name, perform the necessary changes to the configuration files was not feasible anymore. Under these circumstances PMS, such as Red Hat Package Manager and Advanced Packaging Tools, rely on Packages to bundle all this related information into a single file.

2.1.1.1 Red Hat Package Manager

Red Hat Package Manager (RPM) is a PMS developed by Red Hat used in several Linux Distributions. According to [7], to build a package we need to write a Spec File. A Spec File is comprised of eight sections and contains all information and instructions required by the RPM to build a package.

- The Preamble - It summarizes all information associated with the package. For instance, name of the package, version, release, location of the source code...
- The Prep Section - Instructions are provided to make the necessary preparations before the building of the software. As an example, we might need to unzip the software before encapsulating it;
- The Build Section - As in The Prep Section instructions are provided. However, they are utilized to compile the software code;
- The Install Section - Command are specified here to install the software;
- The Files Section – List of files that are part of the package;
- Install and Uninstall Scripts - Contrary to The Install Section, scrips are written in this section to run prior or after the package is either installed or deleted. For example, run configuration files;
- The Verify Script - Script is provided to examine if the package was installed with success, without errors;
- The Clean Section - Instruction can be written in here to clean specific places after the building process is complete.

2.1.1.2 Advanced Packaging Tools

In the same vein as RPM, Advanced Packaging Tools (APT) is a PMS utilized for managing Debian[1] Packages. Following the documentation presented by Debian, to package an application it is necessary to create four files:

- Control - Contains information required to install the package. Namely the name of the package, the architecture supported, dependencies...
- Changelog - When the application is updated, the changes must be specified in the Changelog. A small resume with the changes must be presented with the new version, name, urgency...

[1] https://www.debian.org/

- Copyright - As the name implies, this file contains all information related to copyrights and licenses used in application being packed;
- Rules - Rules File define the target file to which a set of rules that can be applied and specified at the same time each of these rules, such as build, clean, binary...

2.1.2 Containerization

Containerization is a virtualization technique that was developed for the purpose of isolating process resources in the absence of any hardware requirements. Each container can run their own Operating System (OS) but share the same kernel [8]. In other words, it provides an isolated environment specific for the application it encapsulates. This environment solely includes the essential (libraries, OS, etc) for the application to execute. Consequently, this enable the optimization of resources per application and portability between different platforms [9]. Linux Containers (LXC)[2] and Docker are the most utilized technologies that takes advantage of Containerization.

2.1.2.1 Linux Containers

LXC allows the management of containers in Linux distributions through the usage of console commands or through C or Python API. In the interest of packaging and application, LXC accomplish this via Containers Images. A Container Image is an executable package comprising all the essential information (system libraries, tools, etc) to launch a container. The process of customizing a container begins the creation of a container with a base Image, normally of an OS system. Afterwards, it is performed the installation of all software requirements needed to run the application. Lastly, a new Container Image is created from this process. This can be done with a script using the API, relieving the burden of doing manually. The subsequent deploy of this container will be executed through the new Image, instead of all the previous steps. Consequently, time is saved. With the further development of the application and the increasing in dependencies, the Container Image must be updated. The same procedure is utilized, except the base Image will be the previous one instead of the base.

2.1.2.2 Docker

Along the same line, Docker aims to automate the deployment of application into containers, easing the process of packing the application with all its dependencies. Since it builds its solution on LXC techniques [9], its usage is similar to LXC. However, Docker diverges from LXC in one points. First, Docker is not restricted only to Linux distributions, it can run in Windows and macOS. Additionally, the procedure of building a Container Image can be accomplished via DockerFile. DockerFile comprises a series of instructions with its respective arguments [10]. The essential instructions that must be present are:

- From - utilized to specify the base image to operate on;

[2]https://linuxcontainers.org/

- Label - specifies who is the author of the image is and their contact;
- Run - executes the commands followed by the Run instructions on the current image.

2.2 Cloud Deployment

With the application packed, following the CD pipeline, is the Production stage. This is translated in provision a cloud environment to install the application. In other words, a Virtual Machine (VM) Instance must be launched with the desired hardware requirements capable of supporting the application. This can be done via web, console or APIs.

When creating a VM instance via web browser, the user must choose machine configuration by filling a form with personal information and performing a series of clicks, in the desired hardware options.

On the other hand, cloud Providers, as Amazon AWS[3] and Google Cloud Platform (GCP)[4], developed a Command Line Interface (CLI), which allows management of the cloud services. The process of initializing a VM Instance is done through a set of command. In these commands, the specification of the cloud hardware (RAM, Storage Size, CPU, etc) and metadata (User Information) are utilized as arguments. However, provisioning through CLI demands the fulfillment of two requirements. The first requirement is a billing account. The second one is a pair of keys used for the authentication of the user.

Furthermore, some Cloud Software Framework, such as Openstack[5], provide APIs to ease the process of automation of deploying and managing cloud instances. This is done through code compatible with the API, instead of doing the procedure manually. These procedures when done manually can be a nuisance the user. Configuring a machine for each application manually is time consuming. In the long run automatization must be ensured. This can be done with Configuration and Management Tool (CMT) and Server Provisioning Tool (SPT).

2.2.1 Configuration and Management Tools

Configuration and Management Tools (CMT) are tools utilized to automate the management of IT Infrastructures. They provide rapid and smooth modifications to the infrastructure configuration as well as provision. CMT possess a repository to preserve different configurations [11]. As a result, the user can reuse, or simply adapt, configurations to deploy new machines, instead of designing from scratch. Moreover, since every information related to the infrastructure configuration is stored in the same place, it is to keep track of how many machines are active, which processes are they running and what configurations they have [11]. This enables quick changes to multiple infrastructures. New CMT are arriving the market, being Chef[6], Ansible[7] and Kubernetes the most dominant.

[3]https://aws.amazon.com
[4]https://cloud.google.com
[5]https://www.openstack.org
[6]https://www.chef.io
[7]https://www.ansible.com

2.2.1.1 Chef

Chef was developed by Opscode to transform infrastructure into code, that utilizes cooking as a metaphor. Chef utilizes a Client-Server architecture. It comprises a workstation, client, server and supermarket.

The workstation provides an environment for the user to manage their cookbook. A cookbook contains all information related to the working infrastructures. It stores the recipes, templates, attributes Recipes are files written in Ruby language. The configuration of the machines is specified in these recipes. They can also state the templates with the attributes. These templates will be utilized to generate additional files that aid the configuration of the machine.

Once a cookbook is fully customized and ready to be utilized, it is uploaded to a Chef Server. Chef Server function as a hub. It stores the latest cookbooks and supply them to the Chef Clients.

Chef Clients are active machines. Each Chef Client must be registered in the Chef Server. When a new cookbook is sent to the Chef Server, it is the Chef Client duty to fetch it. Consequently, it verifies the differences and install the missing requirements. Chef Supermarket is the location used by the community to share cookbooks.

With regard to provisioning, Chef offer a set of Drivers for various services. Depending on which service might be used, some additional details must be specified in cookbook in order to function. For instance, information about the user account to provide authentication, machine hardware Once this is set, the user specifies which driver he desires and a list of the machines. In this list, for each machine a set of recipes, present in the cookbook, is specified.

2.2.1.2 Ansible

Ansible is an open source CMT developed by Red Hat Inc. In contrast with Chef, Ansible does not use a Client-Server architecture. It operates through SSH connections. As a result, the hostnames or IPs of the machines must be written in Inventory files. Ansible utilizes in Python language to manage the machines configurations. However, the configurations are specified in YAML files, referred as Playbooks. Playbooks consists on a collection of tasks, list of actions, that are going to be executed in a top-bottom order. Each task links internally to a piece of code called module [12].

Ansible presents several modules in their website, including cloud provisioning. In the same way as Chef, additional information must be used with Ansible cloud provisioning. This information is passed as argument to the module in the task. Operations for machine configuration, are specified on tasks after the cloud provision task.

2.2.1.3 Kubernetes

Kubernetes[8] is an open-source system for container orchestration. In other words, it is responsible for the management and deployment of containers clusters according to the workload. Kubernetes utilizes a Server-Client architecture. Kubernetes Master is the central unit, accountable for providing the

[8]https://kubernetes.io/

desired cluster state. To perform this, Kubernetes Master must receive the specification through a YAML file via REST calls or command lines. In the YAML file, the containers comprising a POD and how many PODs I want to run must be specified. A POD is a group of containers in the same network, additionally each POD contains a specification on how to run the containers. Kubernetes Master then communicates with each Kubernetes node to achieve the desired state. With the goal to provision cloud instances and deploy containers, Kubernetes provide two tools, Kops and Kubespray. However, Kubespray requires the usage of Terraform to provide cloud provisioning.

2.2.2 Server Provisioning Tools

As a rule, CMT default to mutable infrastructures. In other words, technologies as Chef and Ansible, were originally developed to configure existing machines. They define the code that runs on each machine [13]. In this manner, faster and safer updates can be performed anytime. Machine provision arrived as an extra feature afterward. On the other hand, Server Provisioning Tools (SPT) main purpose is provision. The term modification in SPT, revolve around deployments of entirely new servers. They do not alter existent server configurations to accommodate new software releases. SPT hold onto immutable infrastructures. At the present, Terraform[9] and Openstack-Heat are the most popular SPT.

2.2.2.1 Terraform

Terraform is a STP created by HashiCorps. Terraform code is written in HashiCorp Configuration Language (HCL) in files with the extension .tf [13]. Whereas Ansible and Chef demand the description of every step necessary to achieve the desired end state of the server, Terraform merely requires the description of the end state. For instance, the user previously deployed 5 server and wishes to launch 6 more, due to an increase in traffic along with a new application version. With Ansible the user needs to create a new playbook. This playbook is utilized to deploy 6 more servers and at the same time keep track of the previous deployed servers and update them. However, Terraform does not require this steps. With HCL, the user just modifies the count to 11 and other fields in the former .tf file utilized. Terraform is aware of the states it originated in the past. In detail, Terraform knows that 5 servers were previously deployed. In the event of these 5 servers being up to date, Terraform simply launch 6 new servers. Otherwise, it terminates the previous servers and provision new ones with the demanded end state.

2.2.2.2 Openstack-Heat

Openstack-Heat (Heat)[10] was developed as a mean to orchestrate provisions to the clouds using Openstack architecture as a base. Heat operates with Heat Orchestration Template (HOT) written in YAML format. In addition, it also has compatibility with Amazon Cloud Formation Templates utilized by Amazon AWS. In the same vein as Terraform, Heat comprises of a Declarative Language. Otherwise

[9]https://www.terraform.io
[10]https://wiki.openstack.org/wiki/Heat

stated, it only requires the specification of the end state of the infrastructure, contrary to provide a list of steps to achieve this end state.

2.3 Web Processing Services

As stated before, in the Introduction, there is a necessity to export EO tools to the web context. To tackle this, the OGC developed an Interface Standard, Web Processing Services (WPS). WPS is a web service that enables the execution of computing processes and retrieval of metadata describing their purpose and functionality [14]. These processes are characterized by receiving inputs, executing algorithms and delivering outputs. The input and output can be of three types:

- ComplexData - doesn't describe a particular structure. Passed values must match with the given format;
- LiteralData - encodes atomic data (scalars, linear units ...);
- BoundingBoxData - are coordinates in the form of an array that represents an area.

In order to enable monitor, execution of processes and retrieval of data, WPS offers the following core functions through HTTP POST and GET requests:

- GetCapabilities - This operation allows the client to request information about which operations the server supports and what processes the server has to offer;
- DescribeProcess - This operation provides a brief description of the algorithm it runs, the inputs it requires and the output it generate;
- GetStatus - retrieve information about the running processes;
- GetResult - retrieve the output generated from the execution of the process.

2.3.1 Web Processing Service Transactional

Web Processing Service Transactional (WPS-T) came to light as the answer to package WPS processes and deploy them in WPS servers at runtime [15]. To achieve this, WPS-T make use of the core WPS interface and proposes and extension to this. The extension consists in adding two new operations, deploy and undeploy process, both requested via HTTP-POST through new endpoints. As explained in the WPS section, the GetCapabilities provides information about which operations are supported by the server. Therefore, GetCapabilites function suffered modifications in order to accommodate the new Operations. This includes the descriptions of each supported schema profile for deployment. It is through the schemas that the WPS-T is able to extract the vital information to expose the WPS processes to the WPS server.

DeployProcess operation receives a DeployProcessRequest and its function is to dynamically deploy a WPS process to the WPS server. The DeployProcessRequest must include the WPS identifier, the version of the WPS, language code, a list of all the process identifiers that the client wishes to deploy (WPS Process Description) and the WPS Deployment Profile.

UndeployProcess operation receives an UndeployProcessRequest and its function is to dynamically undeploy a WPS process. The UndeployProcessRequest has the same attributes as the Deployprocess, except that the WPS Process Description and WPS Deployment Profile are exchanged with the running process id.

2.3.2 CRIM WPS

The Computer Research Institute of Montreal (CRIM) develop a WPS with cloud support to assist the testbed experiments efforts in using a cloud computing environment for Earth Observation (EO) data integrated with OGC web services [16], namely CRIM OpenStack Cloud. CRIM WPS simply modifies the core WPS operation Execute and include a new operation GetCloudIDs. With the purpose of enabling the user to deploy processes into the CRIM OpenStack Cloud, Execute operation was augmented with new parameters as input. These parameters can be summarized into three different types:

- Process Parameters - Related to the Earth Observation data;
- Cloud Parameters - Virtual Machine configuration. In detail, task queue identifier, where it will be published, and both the URL where the process is going to be deploy, executed and the output is going to be stored;
- Docker Parameters - All information that permit CRIM OpenStack Cloud to retrieve the Docker image and deploy as a container. URL for the location of the docker image, name, port and version.

On the WPS specification, the Execution operation is responsible for the execution of the user selected process with the process parameters. While in the CRIM WPS, the Execution operation publish a task with the respective process and docker parameters to the selected cloud via cloud parameters. Forthwith, CRIM OpenStack Cloud is responsible to pull the Docker application package detailed in the docker parameter, launch it and execute the task.

CRIM WPS has available a new operation GetCloudIDs, as mentioned before. It allows the user to get a fully detailed list with all the available clouds and information about these. Enabling the user to fill the cloud parameters when performing a execute operation.

2.4 Thematic Exploitation Platforms

Thematic Exploitation Platforms (TEP) are computing platforms whose intention is to supply the user EO data exploitation, service development and product development. In other words, the user is able to discover, manipulate data, deploy processes into the cloud to perform data transformations and share the final product of this transformations.

Algorithms used to manipulate the data, are developed externally to the TEP's. To launch them in the TEP cloud infrastructures, first they need to be registered. The registration is first done by applying an Application Package to an Application Hub.

The Application Package encapsulate users applications, enabling automatic management and sharing between platforms [17]. In the EO field the information model used to represent the Application Package is the OGC Web Services (OWS) Context Document [18]. OWS Context Document comprises of the following necessary information:

- Application metadata - general details about the WPS process. For instance, title, author, date of creation or update... Application container and resource types - A DockerFile must be specified;
- Input Data - To execute WPS processes input files may be required. The location of these file must be specified, either by value (file content) or by URL;
- Output Data - When the WPS process finish its computation, the output generated must be stored in the indicated location.

In the end of the registration of the Application Package to a Hub, the application is registered in the TEP through the input of the OWS Context Document to a desired WPS server. According to [18], the components in the TEP in charge of deploying and executing processes in the cloud are App Management Client (AWC) and Application Deployment and Execution Service (ADES). AMC is the component that interacts directly with the users. AMS is responsible for forwarding Register and Execute requests to the ADES, querying ADES for existing WPS servers and applications. Once it receives the OWS Context Document with a registration request, it sends all the information in this document to the ADES. On the other hand, ADES contains the active WPS servers. It is responsible for deploying application in the cloud and register it in the WPS server. It also submits the required inputs to the application waiting in the cloud for requests.

2.5 Workflow Chain

The process of transforming data into products is done through EO tools via WPS services. Even though a single process may be enough, multiply transformations might be necessary to produce the desired product. This can be achieved by linking several WPS forming a workflow, Figure 2.1. In other words, we redirect the output of a WPS as an input to other WPS and so forth.

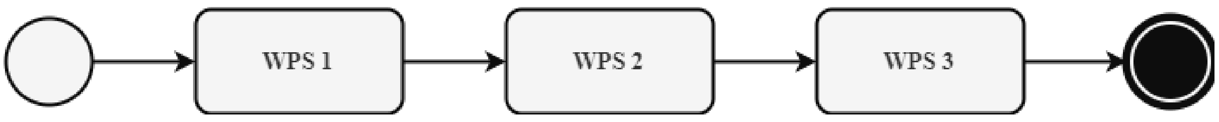


Figure 2.1: Overview of a Workflow Pattern

As an example, we want to create a workflow to identify possible illegal mining activities, Figure 2.2. Given a certain time range, it is possible to observe the activities that occurred in the ground by observing the trend of changes in the minerals. Therefore, our first WPS constituting the workflow is going to take a set of Sentinel-2 L2A images and aggregate all the value for spectral band available from these images for a given region, i.e, Johannesburg, into a single image file. Our second WPS in the chain, is going to take as an input the file containing all the bands and perform a cloud correction. In other words, we perform an algorithm to remove the clouds from the image. At the end, we obtain an image ready for analysis to determine if there were indeed illegal activities.

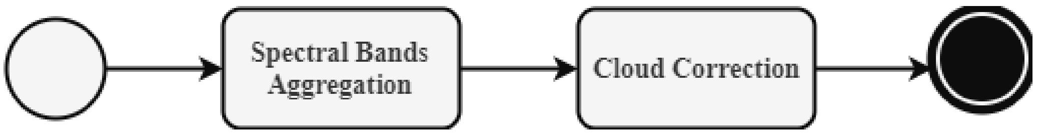


Figure 2.2: Identify illegal mining activities workflow

From the chaining of WPSs, a crucial problem emerged as mentioned in [19], how do we execute a workflow in the cloud environment with enough confidence that it will not fail and waste resources, such as allocation of machines on the cloud under a pay-per-use model?

This can translate this problem into a validation problem. In other words, a failure of a WPS in the workflow chain cause the whole workflow to abort. Thus, resulting in an economic and time loss. The failure of a WPS can result from three possibilities:

- Mismatch between outputs and inputs data type from one WPS to the next WPS in the chain;
- Insufficient data. In other words, not enough data was generated from a WPS;
- Unavailable WPS. In case a WPS is inaccessible, the execution from that WPS onwards will continue to fail.

2.5.1 Workflow Engine for Earth Observation Services

Workflow Engine for Earth Observation Services (WEFEOS) allows the creation, execution and storage of workflows. It presents a friendly graphical interface with a drag-and-drop model. WPSs are represented by boxes, while data types are represented by connectors. In addition, it requires the user to specify the location of the inputs of the WPS. Seeing that WEFEOS offers the possibility to store workflows, they are stored as a template, without the data bindings. This enables to refine and execute the workflow chain with new parameters.

Despite every workflow management system include some of these features, what make WEFEOS unique is his validation. There is no other workflow management system that provides validation of the workflow at the present time. WEFEOS validation can be split into four unique levels. Each level is solving a sub problem of the whole validation on its own [19]. In order to move to the next level, it is required to first must complete the previous one.

In the first level, WEFEOS is responsible to check if the described process of a WPS process specification complies with the standard.

In the second level all data types and restrictions ought to be respected. With this in mind WEFEOS seeks to solve every problem that may arise with each data type. For LiteralData, the input must respect its data type, i.e, be a string, integer, boolean. It must be within the allowed values and if within the range. The following data type, BoundingBox, uses Coordinate Reference System (CRS). Values, defined by longitude and latitude, are check if they are within the bound of the chosen CRS. Lastly, ComplexData the input must abide by the specified mime type and if the mime type is according to its schema, application/json or application/xml. Verify if the input is written in the specified encoding. To finish, the file size must be $\leq$ to the acceptable maximum file size [19]. The natural solution used by WEFEOS comes from the use of the XML constraints in order to build validators. There are four types of validators, where three of them correspond to the three data types used by the WPS. The remaining validator is a Process Validator. It is composed by other validators and presents an operation IsValid enabling to know if the input is valid or not. Consequently, it allows to attach a new validator for every input of a WPS process, solving the problems of this level.

Once third level is reached, the validation starts to resolve around the workflow creation. Internal inputs compatibility and minimum/maximum occurrences are examined. In particular, to not miss any workflow requirements, for example the missing data, the workflow specification is inspected and verified against its schema (JSON). In other words, this schema contains every rule for how the workflow must be specified. This includes the description of the WPS and their fields along with the connections and their direction for the chaining. Through these rules it is possible to observe if everything is correct, the construction, every crucial information is specified by the users and if the WPS chain is compatible.

In the final level, the only missing piece comes down to runtime validation. In specific, being able to validate the output originated by the execution of a WPS before being sent to the next WPS in the chain.

2.5.2 Summary

All things considered, we analyzed several DevOps tools corresponding to each CD pipeline. Beginning with the application packaging, the encapsulation of the EO tool binary, PMS and Containers unique characteristics were described. While PMS automates the process of encapsulating the dependencies of the application and installing them, Containers aim to provide an isolated environment with all the dependencies required for the application to run. APT and RPM were the elected PMS tools, Docker and LXC as the Containers tools.

Following with the cloud deployment we presented in one hand CMT and on the other hand SPT. CMT provide infrastructure as code, enabling its management. Alternatively, SPT focus solely in provisioning cloud instances. For CMT we selected Chef and Ansible to explore. While for SPT we selected Terraform and Openstack-Heat. With each tools specified, we still analyzed other work done in the EO field. To provide exposure of the EO tools to the web context we presented WPS and WPS-T. CRIM WPS and Thematic Exploitation Platform were explored to provide deployment to the cloud.

Equally important to the development of automated CD pipeline, is the monetary costs associated with the usage of cloud platforms. To prevent unexpected errors from input and outputs redirections, WEFEOS was selected.

Chapter 3

Tools Analysis

With regard to the technologies mentioned in the Related Work, several solutions can rise from the combination of these to create the CD pipelines. The focus of these pipeline will be for the system to convey the packaging of an EO tool as well as for the Cloud Deployment to enable the execution of a workflow in the cloud. We will start by examine the strengths and weaknesses of each technology and compare them to one another.

3.1 EO Tool Packaging

With respect to the Build stage of the CD pipeline, namely EO Tool Packaging, we will begin by examining the Package Management Systems (PMS), RPM and APT.

In the first place, with consideration for RPM, there are two main issues. Catching sight of the abilities of PMS, they not only provide support for encapsulating applications, but they also provide support during the installation of these in the target environment. An application might require other applications or libraries installed in order to function. Even though the application requirements are specified when packaging the application with RPM, according to [7] RPM does not perform the installation of these requirements prior to the application. In other words, the user must install them manually beforehand. RPM will only inspect if the requirements are met, deciding afterward if the installation is possible or not. Apart from this, RPM application packages are bound to a specific computer architecture [7]. In the event that the application is pretended to be installed in a machine whose specification is not present in the spec file, it is doomed to fail. Moreover, with the progress in technology, new processors architectures are being develop and released, it is not possible to take into account all of them. Simply put, we can not specify all architectures on the packaging in advance. To allow an RPM application package to work in multiple architectures, either a new package is created or the old one is updated to accommodate the new desired architecture.

Despite the fact that APT does not have these problems, it is locked to Debian operating systems only. With the extended existence of operating systems, APT does not seem an attractive option. Another key point, when analyzing both PMS, is the knowledge and complexity behind each packaging operation.

On one hand we have APT, which comprises of four different files, each one with their specific rules and function. On the other hand, we possess RPM, although it only utilizes one file, this file is composed by eight sections.

In the face of PMS, it is available another technology, designated Containerization, that settle with Application Packaging. Containerization, as explained in the Related Work section, function through containers that offer a unique environment tailor made to each application. The environment only includes the application requirements and dependencies. The major property offered by Containerization is the isolation. Containers are not dependent of any hardware. This means, contrary to RPM and APT, applications will not be restricted specific systems, allowing for portability between them.

While examining with ones eyes, both LXC and Docker, they present a bigger difference, besides the fact that LXC is limited to Linux distributions as opposed to Docker. The difference is in the process of creating a Container Image. The creation of a Container Image, with LXC, is done through the usage of an already running container with all the dependencies of the applications. This procedure can either be done manually, which goes against the main goal of this thesis, or through a script with all the commands necessary offered by LXC API, whether in C or Python.

In contrast with LXC we have Docker. One of the objectives when developing Docker was to ease the management of LXC containers. In spite of needing an script in C or Python, Docker generate an Image through a Dockerfile. A single file with all requirements of the application to create an image. As a result, is less complex than LXC or having a file with eight sections, as RPM, and at the same time faster. Additionally, the existent documentation available for Docker is more than the LXC or PMS. All things considered, the perfect viable solution used that will be explored is the Docker Containers. With Docker, we do not have to worry with the restrictions imposed by the architecture or operating system, as present in both PMS and LXC. In addition, the complexity present by Docker is far less when comparing with PMS and LXC.

In summary, table 3.1 presented below present in a structured manner some of the differences of each tool described.

Tools	***Installation***	***Hardware Architecture***	***Operating System***	***Number of Files***
RPM	Application only	One package per architecture	Linux	One
APT	Application and its dependencies	All	Debian	Four
LXC	Container provides an environment with application and its dependencies	All	Linux	One
Docker	Container provides an environment with application and its dependencies	All	Al	One

Table 3.1: Characteristics of Application Packaging Tools

3.2 Cloud Deployment

With Docker being chosen as the tool to encapsulate the application, the following step is to provide a mean to deploy the container in the cloud, the Production stage. Thus, we analyze the tools presented in the Related Work section.

Since chef honored its craft in configuring already active infrastructures, provisioning as not given much attention. As a result, Chef present some weaknesses in this field. First, Chef utilizes a Client-Server architecture. Hence, when provisioning instances it will be necessary to install a chef agent in this instance. Second, provisioning with chef is done through the usage of drivers developed externally. Therefore, each driver as its own set of rules. An example of this is the use of credentials to provide authentication. While the driver for amazon AWS requires the use of a credential file or a resource file with a path to the keys, other drivers might require specifying the keys directly through the code. Third and last, there is a lack in documentation. Overall, cloud provisioning with chef was designed to be a manual process, adding some complexity when trying to automate it.

In the same vein as Chef, Kubernetes utilize a Server-Client architecture. However, Kubernetes provide a specific tools for cloud provisioning, unlike Chef. As presented in Related Work section 2.2.1.3, Kubernetes provide Kubespray and Kops as tool for cloud provisioning. Thus, we do not need to worry about installing the client on the provisioned machine. Both tools present disadvantages when analyzed. Kops only has available provision to AWS and Google Compute Engine. With the main requirement to deploy to the cloud, Kops will not be useful. Kubespray, in contrast with Kops, require Terraform for cloud provisioning. Therefore, Kubernetes is not a viable option for the task of Cloud Provisioning.

In contrast with Chef and Kubernetes, Ansible does not require a Server-Client architecture. Instead, Python is the only requirement needed to run Ansible. Ansible provision comes from the use of modules in tasks. As stated in the Related Work section, tasks are represented in a YAML file. YAML files have the advantage of being compatible, interactable with every programming language. In detail there are many libraries that support the manipulation of these files. In addition, it is important to hold in consideration, that Chef as well as Ansible, utilize a Procedural approach. In other words, the end state of the VM instance is reach through the application of several steps. Dissimilar to CMTs, SPT such as Terraform and OpenstackHeat, utilize a Declarative approach. The user only specifies the end state of the VM Instance and the SPTs will figure how to provide it.

Terraform is a SPT, developed with the purpose of deployment to the clouds, it has the advantage of having support for numerous providers. Furthermore, Terraform can be extended in case of a provider missing through GO code. Terraform operates with HCL, as mentioned in the Related Work section. This can be a disadvantage, since Terraform only provides full API with the paid version. Terraform Open Source version function through shell commands. Under this circumstance, to use Terraform it will be required the management of shell scripts.

On the other hand, OpenStackHeat provide RestAPI and CloudFormationcompatible Query API, which is compatible with every code. As Ansible, OpenStackHeat has the advantages of using both Python with YAML file templates. The main problem of OpenStackHeat comes from the fact that is restricted to OpenStack. Even though it supports CloudFormation templates, it cannot provision in Amazon AWS.

In the final analysis we will explore a solution built upon Ansible. When comparing Ansible with OpenStack-Heat, both use similar approaches. However, Ansible possess a wide variety of providers in contrast with OpenStackHeat. The same justification can be utilized for Kubernetes Kops, since it only has compatibility with AWS and Google Cloud Engines. In relation to Terraform, the full power of it comes from paid version. For the free version, it would require the managing of script. Ansible is an open stack SPT provides a Python API, releasing us from this burden. Therefore, Ansible has the same advantage in comparison with Kubernetes Kubespray that requires Terraform too. At last, with respect to chef, Ansible seems the superior choice. In the same way as Ansible, Chef offers a Ruby API. Howbeit, Chef requires additional steps since it is based on a Client-Server architecture. Moreover, Chef driver for provisioning are lacking when compared to Ansible. Ansible developed provisioning further. Given these points, summarized in the table 3.2, Ansible has the upper hand compared to the other tools, to help us solve the problem of automate the process of deployment to the cloud.

Tools	***Client-Server Architecture***	***API***	***Business Model***	***Cloud Providers***	***Templates***
Chef	✓	Ruby	Open Source	All	-
Ansible	-	Python	Open Source	All	YAML
Kubernetes	✓	REST	Open Source	AWS and Google Computing Engine	-
Terraform	-	REST	Enterprise	All	HCL
Openstack	-	REST	Open Source	Openstack	YAML

Table 3.2: Characteristics of Cloud Provisioning Tools

3.3 Web Processing Services

As presented in Related Work section 2.3, WPS was developed by OGC as a standard to render the EO tools to the web. Being widely accepted, developers started to share EO tools through WPS servers. However, numerous EO tools still require to be translated to WPS. Despite TEP allowing the registry of these EO tools in a WPS server, it will not be taken into consideration since TEP only provides a graphical interface.

In the interest of offering EO tool via WPS, we came across two possible solutions. The first solution consists in the usage of WPS-T. WPS-T server allows to dynamically register an EO tool via HTTP/POST method. Although seeming the perfect answer to our problem, WPS-T is still in development and only start to arise recently. Under these circumstances, WPS-T may still be prone to unexpected errors. With the deployment to the cloud, unforeseen faults will result in a monetary loss, since all progress will be lost. It is essential to avoid these problems. Nevertheless, WPS-T is a solution which will be held into consideration and explored.

The second and more stable solution is the use of templates for the WPS. The source code, types of inputs/outputs and metadata will be the major attributes specified in the templates, to accommodate each EO tool. Therefore, in spite of dynamically register the EO tools into the WPS server, we deploy the WPS specified with already the predefined EO tools. Hence, preventing unexpected errors. As a result, we will add a new stage responsible for the encapsulation of the EO tool into a WPS standard, in our CD pipeline after committing the code.

With special attention to minimizing the costs, fault prevention becomes vital. The main source of errors came from the linkage of WPSs, creating a workflow chain. The development of Workflow Engines (WE) aimed to relieve the users from the process of executing each WPS manually. Despite the automation, the user is still required to ensure each output and input is valid, otherwise the execution will fail. Henceforward, we will use WEFEOS presented at Related Work section 2.5.1. WEFEOS not only offers a graphical interface that enables the creation, execution and share of workflows, compatible with every engine, but also syntactic validation at several levels. As a result, it is a crucial component to utilize.

With WPS server being chosen as the technology to provide EO tools over the internet, there still exists two possible approaches to deploy them to the cloud. The first consists in deploying only the process that will be running. Both TEP and CRIM WPS utilize the last procedure, with small differences. CRIM WPS proposes a modification to Execution operation in the WPS standard. As specified in Related Work section, new parameters are attached, allowing to deploy to the cloud. On the other hand, TEP does not adjust the WPS operations to deploy the EO tool to the cloud. Instead, TEP, upon receiving a registration of an EO tool, will straightforward deploy the EO tool to their cloud waiting for requests. In other words, TEP forwards Execution operations to the EO tool already running in the cloud. Alternatively, in spite of deploying solely the EO tool, WPS can be deployed as a whole to the cloud infrastructures.

In the final analysis, the deploy of the EO tool may seem the right option at first. However, as can be seen this option requires the modification of the standard defined by the OGC. Systems that require the standard interface would not be viable anymore, i.e WEFEOS or others WEs. In the end, we would lose interoperability. Therefore, the viable option to be explore will be encapsulate WPS and deploy it to the cloud. All things considered, we will use WEFEOS as the base for this thesis to provide both creation and validation of workflows. We will explore the use of templates to accomplish the wrapping of an EO tool into a WPS. With the help of the Docker and Ansible, we will deploy the whole WPS to the cloud, alternatively to perform modifications to the OGC standard.

Chapter 4

Implementation

In the wake of defining the technologies required for the application packaging and cloud deployment automation, we now present each technology used to solve both challenges. In other words, we will explain our solution in this chapter. Thus, we will start up with a global overview of the system, by take into consideration the core modules and their accountability. Subsequently, we will progress deeper into detail about the achievement of the full automation of both pipelines through the use of the Multi-Cloud Deployment and Execution of Earth Observation Service (MCDEEOS). Lastly, we will explain how the users interact with our system via MCDEEOS-WEB features. Along with both modules, we explain how they utilize WEFEOS to perform the necessary validations.

4.1 System

As mentioned in the previous chapter, we will be using as the base for this thesis WEFEOS. WEFEOS was tested using WPSs to identify and map wildfire events in the real-world. It was based on the Copernicus workshop examples. The battery of tests implemented the following test cases:

- Utilize wrong values;
- Simulating data type mismatches;
- Missing data;
- Insufficient data.

With the tests cleared, meaning WEFEOS detected every error, it exhibits a strong stability. We consider WEFEOS to be the right choice as the base. WEFEOS system is comprised by WPS-V-WEB and WPS-V.

WPS-V-WEB is responsible for the workflow management in the web browser. It has available a dashboard with a menu. In here, the users are able to compose, download and load workflows. During the composition of the workflows, it alerts the users of the compatibility issues between WPS processes. In other words, if the outputs match the specification of the input of another WPS process. Nevertheless, WPS-V-WEB also provides a console to display the errors of the input data specified. An example of a possible error is the mismatch between the image type the WPS is expecting to receive and the image type actually specified.

To detect these errors, it is required a syntactic validation available with the use of WPS-V. Furthermore, WPS-V provides dynamic validation for the output generated by each WPS process. As a result, our solution incorporates these core modules, as can be seen in the Figure 4.1.

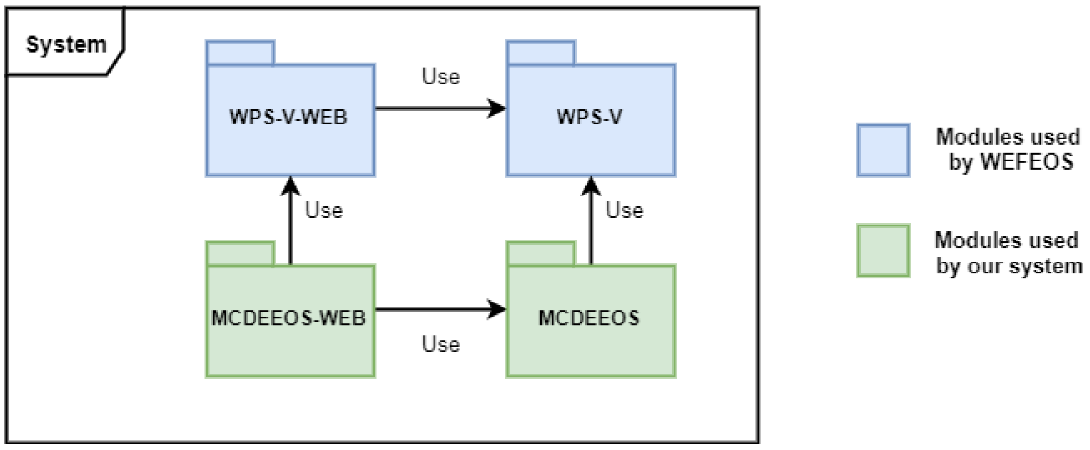


Figure 4.1: List of WPSs present in a catalog in the system

MCDEEOS is the module responsible for providing the EO Tool Packaging. It is through this module that the EO tool is wrapped into a WPS, dockerized and stored in a repository for later usage. In addition, MCDEEOS is also responsible for forwarding and preparing, if necessary, the workflow to a workflow engine for its execution, as well as to provide deployment of the WPS's to the cloud, validation of the outputs by using WPS-V-WEB and their termination. To offer these functionalities to the users we developed the MCDEEOS-WEB. MCDEEOS-WEB is responsible for providing a web interface for the users to specify all the information necessary to perform both application package and workflow execution on the cloud.

4.2 Security

Prior to going deeper in detail concerning our modules, it is necessary to remind security was not one of our concerns. Despite that, it was implemented some basic security measures. To access the platform, users are required to register first. Through this registration, critical personal information, password and cloud secret keys, are specified and stored in a database. However, they first receive some transformation before being stored.

Passwords are only used for the initial authentication, the login. There is no further use to them, as a result, we hash them and store the result from this hashing process.

Cloud secret keys, on the contrary, are required for each deployment the users want to perform to provide access the cloud features. In this situation, these keys are ciphered with a symmetric key only known to the system and then stored in the database. Each time they are required to be used, we just need to decipher first.

With regard to user authentication, we do not want for the users to send the username and password with each request. This is not practical as well as it is more prone for the possibility to disclosure the password, which the user may use in other services. Therefore, we utilized JSON Web Tokens (JWT). JWT can be characterized as a string with the following format header.payload.signature. The header contains the information on how the JWT is computed, i.e, the algorithm utilized. The payload include data about the entity along with additional information. Both the header and the payload are encoded with base64 to form the JWT. Finally, the signature is comprised of the hashing resulted from the junction of the encoded header, encoded payload and a secret. This allows to verify if any of the JWT suffered unwanted modifications. As a result, when perform an application package our cloud deployment in each request it is included the JWT to provide authentication. It is important to bear in mind we are using HTTP as the protocol for the communication. Considering the channel is not ciphered, we are susceptible to eavesdrop and therefore, to personification attacks. To prevent this, the communication should utilize HTTPS instead. Even though the change would be accomplished in minutes, it would require a paid certificate. However, as previously mentioned, security was not one of our requirements during the development of the system.

4.3 MCDEEOS

We will proceed to explain how we were able to accomplish our goals of packaging an EO tool and deploy a WPS for the workflow execution in our core back-end package MCDEEOS.

4.3.1 EO Tool Packaging

As previously explained, the application package pipeline has three stages. We will now describe which challenges we meet during the development of each stage in the application packaging and how we coped to solve them.

4.3.1.1 EO Tool Wrapping

The first stage corresponds to the wrapping of an EO tool into a WPS. In other words, we need to develop the standard interface of a WPS for retrieving metadata about the numerous processes it contains and their execution through the methods described in chapter 2.3. One possible solution for this is to automatize the wrapping process by creating all this REST endpoints for each EO tool. However, this is not practical. The GetCapabilities and DescribeProcess all retrieve an XML document with all the information regarding its EO tool and processes. An example of this XML file is present at Listing A.1 in Appendix B. The creation of these endpoints to deliver the XML documents can reveal to be a complex task and difficult to avoid errors. It is crucial that these documents are well formed with every detail about the EO tool and according to the standard given that WEFEOS use them to perform its syntactic validation. As a result, instead of developing the endpoints from scratch, we searched for an already existent open source WPS implementations. Thus, we found and decided to utilize a python WPS implementation named PyWPS, avoiding reinventing the wheel.

To use PyWPS as the base for turning the EO tool into a WPS, we first analyzed which files it required to change in order to accommodate for each tool. As a result, we identified a bare minimum of two files, since each WPS has at least one process. In other words, we need one file for each process and another for the main file. The Listing 4.1 shows a simple example of a python class for a process file that says hello. This file can be seen as five pieces. The first corresponds to the imports, where the required libraries are specified. The second is the name of the class which corresponds to the name of the process. The third is the use of a list containing functions specifying each data type for all inputs and outputs. In this example, the process is expecting a LiteralInput, a string with the users name, and a literal output of type string. Fourth is the specification of the metadata about the process. Lastly, it is required a handler function for specifying what is the result obtained from the execution of the process. In this illustration, the process will return a string saying hello with the uses name.

```python
from pywps import Process, LiteralInput, LiteralOutput, UOM


class SayHello(Process):
    def __init__(self):
        inputs = [LiteralInput('name', 'Input name', data_type='string')]
        outputs = [LiteralOutput('response', 'Output response', data_type='string')]
        super(SayHello, self).__init__(
            self._handler,
            identifier='say_hello',
            title='Process Say Hello',
            abstract='Returns literal string output with Hello plus the inputed name',
            version='1.3.3.7',
            inputs=[LiteralInput('name', 'Input name', data_type='string')],
            outputs=[LiteralOutput('response', 'Output response', data_type='string')],
            store_supported=True,
            status_supported=True
        )

    def _handler(self, request, response):
        response.outputs['response'].data = 'Hello ' + request.inputs['name'][0].data
        response.outputs['response'].uom = UOM('unity')
        return response
```

Listing 4.1: SayHello Class of a WPS

To replicate such file, tailored for each process of the EO tool, we use a pre-defined template, 4.2.

This template use variables, identified be the use of {{ }}, that we are able to change when the users request an application wrapping. An example of these is the {{class_name}}, where in our code we use class_name as the identifier for the class name that will replace it when the document is created. As observed in the Listing 4.2, templates can also include loops.

```
{%- for lib in imports %}
    {%- if loop.first %},{% endif %}
    {{- lib}}
    {%- if not loop.last %},{% endif %}
{%- endfor %}

class {{class_name}} (Process):
    def __init__(self):
        self.wps_name = ''
        inputs = {{inputs}}
        outputs = {{outputs}}
        super({{class_name}}, self).__init__(
            self._handler,
            identifier="{{id}}",
            title="{{title}}",
            abstract="{{abstract}}",
            version="{{version}}",
            inputs=inputs,
            outputs=outputs,
            store_supported=True,
            status_supported=True
        )

    def _handler(self, request, response):
    {%- for ivar in input_vars%}
        {{ ivar}}
    {%- endfor %}
        process = subprocess.check_output({{shell_command}})
    {%- for rf in move_files%}
        {{ rf}}
    {%- endfor %}
    {%- for ovar in output_vars%}
        {{ ovar}}
    {%- endfor %}
        return response
```

Listing 4.2: Process Class Template

These loops are represented in the same fashion as any other programming language. The use of a loop in the first section is for the specification of the several libraries, since what we want to import depends on the input and output data type. Moreover, prior to explain the usage of loops in the last section , it is important to notice that in Listing 4.1, the code for process is written in the file itself as in our case this is not possible. Rewriting the code in the process file is not feasible when automatized. To accomplish this, it would require having knowledge on the processes of each EO tools and be able to parse them as well as the result accordingly. To solve this problem, we use a library that enable us to call external commands. In other words, we are able to use the normal console command to execute the EO tool process with this library. Considering the example above, with the difference that the say hello code would be written in a regular python file, through the library we are able to evoke this program with the following command python3 SayHello.py. These EO tools processes can receive multiple arguments for their execution. With the example of the say hello, the users might pass their name as input. As a result, we use the first loop to state as variables for each input specified by the users and pass it to after the command line. PyWPS utilize an endpoint for exposing the results obtained from executing of a process. It is important to state, all outputs obtained from the EO tools are ComplexData, i.e, they originate files as results. This exposure is done from a folder named outputs. As a result, we need to move the resultant files to the outputs folder. The location of the results is specified by the users, since we cannot obtain this information through other method. Therefore, the second loop is utilized for moving each output file to the outputs folder. At long last, we need to specify the results. Although, we obtain files, we do not specify their content, instead we specify its location. The second file that we need to change is the main file. It is in the main file that all the endpoints are specified. However, we only need to change a code line of the file, Listing 4.3. In the processes list we instantiate the object for each process class. The same process as described above for the process file, using a template, is applied in here.

```
app = flask.Flask(__name__)
processes = [
    SayHello(),
]

process_descriptor = {}
for process in processes:
    abstract = process.abstract
    identifier = process.identifier
    process_descriptor[identifier] = abstract

# This is, how you start PyWPS instance
service = Service(processes, ['pywps.cfg'])
```

Listing 4.3: Snippet of the code from the Main Class containing the SayHello Process

Furthermore, during the development of MCDEEOS a new requirement arose. With the EO tool being executed in the cloud, the users would not have access to the console terminal. As a result, error messages threw by the WPS during requests would not be visible. PyWPS already has an inbuilt folder names logs for this case. However, to access it we augmented with an extra endpoint for this in the main file.

4.3.1.2 Web Processing Service Encapsulation

With the wrapping of the EO tool finished, we move to the next phase of the pipeline, the encapsulation process. For this, we opted to utilize Docker since it is the most flexible technology when compared with the others, chapter 3.1. At the same time while reading the OGC engineering reports we observed Docker was the tool chosen for the encapsulation by the community. It is important to mention the EO tool is required to be available at the Dockerhub in order for us to have access to it. With this we have a deeper level of insurance that the EO tool is going to work, since the Docker environment is already tailored for the tool. Docker utilizes a powerful layering system. In other words, when building images, it can create several layers during the process. This is similar to checkpoints, enabling the use of a given layer to build a new image from it. As a result, we use the Docker images containing the EO tool as the base image for the encapsulation of the WPS. Thus, the Dockerfile for the image building is going to differ from each EO tool. To create this Dockerfile we used once more a template, Listing 4.4 . In here, we use the variable image to provide the EO tool Docker image along commands to create and transfer the WPS files to it and install all its dependencies. It is important to notice the last line of the Dockerfile. As mention previously, PyWPS has a logs folder that has access to a database where all the details about the errors are stored. However, with the execution of this line we are redirecting the console log to a text file, in case users do not wish to go through the database.

```
FROM {{image}}
WORKDIR /wps
COPY . /wps
RUN apt-get -y update && \
    apt-get -y install python3 && \
    apt-get -y install python3-pip && \
    apt-get install git -y
RUN pip3 install pywps==4.2.1 && \
    pip3 install Flask
EXPOSE 5000
CMD python3 demo.py > ./logs/logs.txt 2>&1
```

Listing 4.4: Dockerfile Template

Moreover, Docker provides a REST API easing its integration in our system. As a result, we use HTTP Posts and Gets to manage everything Docker related. As an illustration, we use this for building of the Docker image containing the WPS by using the Dockerfile created by the above template.

4.3.1.3 Retrieval of Web Processing Services XML Description

During the development of our system we discovered that WEFEOS was designed with the objective of execution workflows locally instead of the cloud. This rose a problem in which WPSs must be already running in order to validate their syntactics. A solution for this was the deployment of the WPS first locally, before deploying in the cloud. However, with the possibility of several requests being made, this is not feasible. To solve this problem, when the users perform the application packaging in our system, we do an initial deploy and through this, we query the WPS with a GetDescription and save the its specification on a XML file. Every time WPS-V requires to validate the syntactic of a WPS instead of query the WPS itself it will query our system through an endpoint to obtain the designated XML file. As a result, a new stage arose in the pipeline, after the Docker encapsulation and before the storing the image in the repository, Figure 4.2.

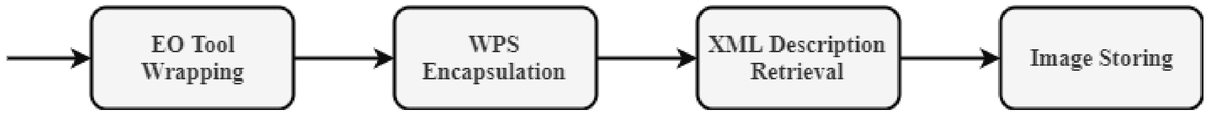


Figure 4.2: EO Tool Packaging CD Pipeline

4.3.1.4 Web Processing Services Image Storing

The process to the storage of the WPS image was rather straightforward. Since we opted to use Docker as the technology for the encapsulation, we choose to use Dockerhub as the repository. As result, Docker already provides all the commands to store and manage images in the Dockerhub through the REST API, solving this stage without any difficulty or challenge.

4.3.2 Workflow Execution in Cloud

With the goal to deploy WPS in the cloud in order to execute them in a chain, our first priority was to begin with a simple WPS deploy. In other words, bring to life the second pipeline by provisioning, configuring and deploying a WPS. To achieve we used Ansible. As mentioned in chapter 3.2, Ansible is able to solve the three challenges of the pipeline stages. Moreover, comparing with the others CMTs, Ansible is the tool with more documentation along with a larger set of cloud modules. Ansible utilizes YAML templates to execute their modules. For each stage of the pipeline templates were created. We are going to present the challenges that arose during the development of each stage.

4.3.2.1 Cloud Provisioning

Considering Copernicus DIAS cloud providers had not fully developed their API in time, we search for another viable cloud provider that has access to Copernicus Sentinel data. Through this search, we found AWS meet this criterion, making it the best choice since it is not required to move data between infrastructures. As a result, we began by using AWS EC2 instances. With the use of AWS, Ansible has available an official module for managing EC2 instance. To simply put, we just compelled the crucial tasks together, available in this module, in a YAML file, originating a playbook, in other words a template. We will further describe, step by step, each task composing this playbook.

It is important to bear in mind, our system provision instance in the users AWS account. Therefore, users are forced to provide our system with AWS secret and access keys to enable the provisioning. Along with the keys, the region needs to be specified by the users where the provision is going to occur.

To enable the provisioning in AWS some configurations must be done prior to the provision task. To have access to the cloud instance, it is required the use of an SSH key. This key varies from region to region, Listing 4.5. To accomplish this Ansible has an EC2 module that enables this creation. Upon creating the key, it is required to save it for later use, Listing 4.6.

```
name: Create a new EC2 key pair
ec2_key:
  aws_access_key: "{{ access_key }}"
  aws_secret_key: "{{ secret_key }}"
  region: "{{ region }}"
  name: "{{ ssh_key_name }}"
register: ec2_key
```

Listing 4.5: Create AWS EC2 key pair

```
name: Save private EC2 key
copy:
  content: "{{ ec2_key.key.private_key }}"
  dest: "{{ destination }}"
  mode: 0400
when: ec2_key.changed
become: joao
```

Listing 4.6: Save AWS EC2 private key

It is important to mention, even though we run these tasks multiple times for the same account and region, Ansible has the ability to detect if the criterion of the tasks have been met, skipping the task if they were. In other words, if the key has already been created, Ansible is able to detect and skip this procedure avoiding the unnecessary waste of time creating a key that already exists and can be used. Another example for this event is the cloud instance already having Docker installed. If there is a task stating the installation of Docker in a cloud instance, Ansible would detect and skip it. Every cloud instance must have an IP associated in order to be connected to a computer network. The IP of a cloud instance is compelled in three pieces. The first identifies the Virtual Private Cloud (VPC) the second the Subnet inside the VPC and the third the machine. Therefore, It is required to create a VPC, Listing 4.7 along with a VPC Subnet in the given region, Listing 4.8.

```
name: Create VPC
ec2_vpc_net:
  aws_access_key: "{{ access_key }}"
  aws_secret_key: "{{ secret_key }}"
  cidr_block: 200.20.0.0/16
  name: wps_aws_vpc
  region: "{{ region }}"
register: wps_aws_vpc
```

Listing 4.7: Create AWS VPC

```
name: Create subnet
ec2_vpc_subnet:
  aws_access_key: "{{ access_key }}"
  aws_secret_key: "{{ secret_key }}"
  state: present
  vpc_id: "{{ wps_aws_vpc.vpc.id }}"
  cidr: 200.20.20.0/24
  region: "{{ region }}"
register: wps_aws_subnet
```

Listing 4.8: Create AWS VPC Subnet

The assignment of the last bits of the IP used to identify the machine is done dynamically by AWS. In other words, we do not specify it. It is possible to create static IP addresses. However, it would require our system to keep track of the already in use IP addresses, which we would not benefit from. Although the VPC is already specified, the VPC is still a private network not ready for use. For the execution of WPS they require access to the Internet. Therefore, it is essential to create an Internet Gateway, Listing 4.9 and associate it to the VPC along with setting up a route table, Listing 4.10, to enable the connection of the Subnet to the Internet. Moreover, for users to access the WPS through web browser, it is required to create a Security Group that allows HTTP traffic, Listing 4.11.

```
name: Create AWS internet gateway
ec2_vpc_igw:
  aws_access_key: "{{ access_key }}"
  aws_secret_key: "{{ secret_key }}"
  vpc_id: "{{ wps_aws_vpc.vpc.id }}"
  region: "{{ region }}"
  state: present
register: igw
```

Listing 4.9: Create AWS internet gateway

```
name: Set up subnet route table
ec2_vpc_route_table:
  aws_access_key: "{{ access_key }}"
  aws_secret_key: "{{ secret_key }}"
  vpc_id: "{{ wps_aws_vpc.vpc.id }}"
  region: "{{ region }}"
  tags:
    Name: Public
  subnets:
  - "{{ wps_aws_subnet.subnet.id }}"
  routes:
  - dest: 0.0.0.0/0
    gateway_id: "{{ igw.gateway_id }}"
register: public_route_table
```

Listing 4.10: Create AWS route table

```
name: Create security group
ec2_group:
  aws_access_key: "{{ access_key }}"
  aws_secret_key: "{{ secret_key }}"
  name: "ssh_ssh_group"
  description: sg rules
  region: "{{ region }}"
  vpc_id: "{{ wps_aws_vpc.vpc.id }}"
  rules:
  - proto: tcp
    ports:
    - 22
    cidr_ip: 0.0.0.0/0
  - proto: tcp
    ports:
    - 80
    cidr_ip: 0.0.0.0/0
register: ssh_group
```

Listing 4.11: Create AWS security group to allow HTTP and SSH

At long last, with the configurations done we only require one more piece of information before specifying the provisioning task, the image id. Since the image ids changes from region to region, we needed to find a solution to obtain these ids dynamically. Ansible has an official module, ec2_ami_facts, that enables us to perform a query to obtain information about images, where ids are specified in it. This module allows us to specify the region, the owners and use filters, for example the name of the image, to obtain the desired image, Listing 4.12. With every preparation done, we can now compel it to specify every parameter in the provisioning task, Listing 4.14. At the end of provisioning, we have to store it along with the key location and user in the Ansible Inventory in order to have access to it afterwards, Listing 4.13.

```
name: Get image for the instance
ec2_ami_facts:
  aws_access_key: "{{ access_key }}"
  aws_secret_key: "{{ secret_key }}"
  region: "{{ region }}"
  owners: "{{owner}}"
  filters:
    name: "{{name}}"
register: img
```

Listing 4.12: Get Ubuntu image

```
name: Save the IP to access it
add_host:
  name: "{{ item.public_ip }}"
  groups: "{{group_name}}"
  ansible_ssh_private_key_file: "{{key}}"
  ansible_user: "ubuntu"
with_items: "{{ ec2.instances }}"
```

Listing 4.13: Save the AWS EC2 cloud instance IP to access it when required

```
name: Provision a AWS EC2 instances
ec2:
  aws_access_key: "{{ access_key }}"
  aws_secret_key: "{{ secret_key }}"
  key_name: "{{ ssh_key_name  }}"
  region: "{{ region }}"
  instance_type: "{{ instance_type }}"
  image: "{{ img.images[0].image_id }}"
  wait: true
  exact_count: 1
  count_tag:
    Name: "{{ name }}"
  instance_tags:
    Name: "{{ name }}"
  assign_public_ip: yes
  vpc_subnet_id:
    "{{ wps_aws_subnet.subnet.id }}"
  group_id: "{{ ssh_group.group_id }}"
register: ec2
```

Listing 4.14: Provision a AWS EC2 instances

4.3.2.2 Cloud Configuration

With the cloud provision accomplished, it is now required to devise a template to perform configurations on the cloud instance. Considering that we used Docker to encapsulate the WPS, it is mandatory to install it and pull the WPS Docker image in the cloud instance. Along with the WPS Docker image, we will also pull a Docker image that contains the WPS-V. While the syntactic validation is performed locally, the dynamic validation is performed on the cloud instance instead. By performing the validation of the outputs generated by the WPS execution on the cloud instance we avoid the need to transfer these outputs to the local machine. As a result, we encapsulated the module responsible for the validation, WPS-V, in a Docker image and just deploy it along with the WPS. Despise WPS-V being deployed along with the WPSs, Docker containers are isolated from each other, the data still needs to be transferred from one container to the other. To avoid this, we created a volume, mount it to the outputs folder of the WPSs and attach it to the WPS-V container. As a result, we are able to share data between containers, allowing the direct access by the WPS-V container without the need to transfer it.

It is important to note, with the deployment of the Docker images in the cloud instance, they are not available to the outside world. Simply put, they are running in a local network created by Docker. To solve this, we used Nginx. Nginx is a HTTP server that provides proxy redirection. This allows the requests performed to the cloud instance to be redirected to the Docker container running either the WPS or the WPS-V. To enable the proxy redirection for our needs we need to provide a configuration file. To differentiate the path of the validator from the WPS we added a tag to the location. In other words, if our cloud instance has the ip 192.3.2.1 to access the WPS named wps_crop, we must access it via 192.3.2.1/wps_crop/. As a result, since the WPS name change from WPS to WPS, we use a template to create this configuration file, Listing 4.15 . It is important to denote, the proxy_pass is the flag indicating to which ip running inside the cloud instance we are redirecting the traffic. Upon creating the configuration file, we just move it to the cloud instance, create a symlink so the Nginx knows there is a new configuration available and restart it.

```
server{
        listen 80;


    location /validator/{
                proxy_pass "http://10.10.0.2:2911/";
                proxy_set_header X-Real-IP $remote_addr;
                proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        }


{% for wps_name, ip_addr in wps %}
        location /{{ wps_name }}/{
                proxy_pass "http://{{ ip_addr }}:5000/";
                proxy_set_header X-Real-IP $remote_addr;
                proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        }


{% endfor %}
}
```

Listing 4.15: Nginx Configuration Template

At the end of developing the cloud deployment, we noticed the time spent on provisioning and configuring one cloud instance was too long. To have a better perspective, we have collected the time spent on ten provisions and configuration for three different regions on AWS. As can be seen, based on the Table 4.2, the average time spent for the thirty provisions and configurations is 376,289 seconds. To reduce the time, we came across two possible solutions. The first one was the use of AWS Lambda. AWS lambda enables the execution of serverless code. In other words, whenever it is required to execute a WPS we would trigger an event to load and run the code without using a server. As a result, it would speed the process. However, AWS Lambda has several restrictions, being the major problems the space limitation to five hundred megabytes and the limitation of the usage time to fifteen minutes. Upon analyzing the Docker images of the WPSs we found it surpasses the limit as well as its execution can take hours to complete. Therefore, AWS Lambda did not fit our needs. The second possible solution is to create an image used when provisioning the cloud instance. This image contains the Docker, Nginx and the WPS-V image, since they are required in every cloud instance. With this approach, we avoid wasting time in the configuration along with pulling the WPS-V Docker image from the Dockerhub and instead we are only required to send the Nginx configuration file and execute a task to create a WPS-V container. To test this, we developed the YAML files containing the tasks and repeated the same 10 provisions and necessary configurations for same three different regions on AWS, obtaining the Table 4.1.

As a result, the average time for the thirty provisions and configurations is 197,951. We were able to speed the process by 47,4%. Although, it still requires some minutes to provision and configure, this time is normally masked by the execution of WPSs considering it might take hours. Thus, we opted to use this solution.

To have a better understanding about the time being masked with the execution time, we used three WPSs to calculate Rainfall Anomalies. To calculate the Rainfall Anomalies, we first need to obtain a . An LTA is simply the arithmetic average of a given data set. This data set is composed by several images from a given range of days, months and years. It is necessary at least ten days for two years to calculate an LTA. The calculation of the LTA is for each year specified in the data set. With the same data set, we calculate the total sum of the rainfall for each year of the data set. In other words, we perform several aggregations in this data set. With both the LTA and the aggregations, we determine a typical annual average variable, which is translated in an image for each year. From this image, it is possible to verify if there was indeed a Rainfall Anomaly for the given year. Each stage of the calculation of the Rainfall Anomaly corresponds to a WPS. It is important to mention the information gathered from only two years is typically not used due to the lack of details available. Given this, for 6 years, the time spent processing this chain of WPS is an hour. As can been seen, the average time for deploying a WPS is minuscule compared to the total time spent on the chain execution and therefore masked by it.

Northern Virginia	Frankfurt	Seoul
258.990	207.457	230.638
219.258	173.410	211.313
197.480	154.715	196.172
203.343	144.458	207.182
216.742	145.002	213.250
205.894	141.524	200.590
205.031	144.688	203.020
250.162	143.186	222.693
190.875	143.171	305.252
209.424	143.944	249.686

Table 4.1: Deployment time using self made image in seconds

Northern Virginia	Frankfurt	Seoul
294.313	251.215	465.584
350.050	262.310	458.914
361.268	298.986	450.440
312.530	350.084	442.678
291.758	550.338	470.924
342.600	265.020	464.509
351.427	302.237	489.514
343.080	263.739	469.057
352.961	457.285	483.932
344.278	283.619	464.025

Table 4.2: Deployment time using base ubuntu image in seconds

4.3.2.3 Augmenting the system with more cloud providers

It is important to mention, our system is not bound to AWS as the only cloud provider. As already mentioned, our main focus was the DIAS cloud providers. Despite their API was not ready for release, we design our system with the desire to enable the addition of cloud providers in the future. As a result, to verify the difficulty of augmenting the system with diverse cloud providers, we decided to add Microsoft Azure to our system. Considering that there were no more cloud providers available with access to Copernicus Sentinel data, we decided to use Microsoft Azure since it is a well-known cloud provider.

To increase our system with Microsoft Azure, we had to develop new playbooks for Ansible. It is important to mention, the playbook only has tasks for provision since the configurations are on separate playbooks. The tasks used for Microsoft Azure were similar to the AWS provisioning. Despite the names of the modules changing from one to the other as well as some parameters, the main differences were in the introduction of a network interface, Listing 4.16, which replaces the AWS gateway and the need for configuring routing table, and the creation of a public dynamic IP, Listing 4.17, which AWS assigns automatically.

Overall, the implantation of Microsoft Azure was achieved in two days. However, more time was spent due to the previous versions of Ansible 2.8.0 having some bugs disabling the provision of instances and the learning how to obtain the crucial keys to enable the provisioning through Ansible.

```
name: Create NIC
azure_rm_networkinterface:
  subscription_id: "{{ subs_id }}"
  client_id: "{{ client_id }}"
  tenant: "{{ tenant }}"
  secret: "{{ secret }}"
  resource_group: "{{ res_group }}"
  name: "{{ nic_name }}"
  virtual_network: wps_azure_vpc
  subnet: wps_azure_subnet
  public_ip_name: "{{ pub_ip_name }}"
  security_group: "{{ sec_group }}"
```

Listing 4.16: Create Azure network interface task

```
name: Create public ip
azure_rm_publicipaddress:
  subscription_id: "{{ subs_id }}"
  client_id: "{{ client_id }}"
  tenant: "{{ tenant }}"
  secret: "{{ secret }}"
  resource_group: "{{ res_group }}"
  allocation_method: Dynamic
  name: "{{ pub_ip_name }}"
```

Listing 4.17: Create Azure public IP task

4.3.2.4 Disk Size Modes

Until this point, we have been using the default disk space specified by the different cloud providers. However, this disk space may not be enough for the execution of the WPS. Considering the disk space required changes from WPS to WPS, the best approach is to ask the users how much space it is required. However, we provide two different choices when the users specify the disk size.

The first choice consists in observing the value provided as the possible minimum value required and instantiate with the maximum value available for the instance type that meets every requirement. For example, a user specifies he requires 20 Gigabytes (GB) of space. We analyze at the requirements of the WPS and find that it meets the AWS tier two micro instance. Tier two micro instance disk space can go up to 2 Terabytes (TB). Instead of provisioning with the 20GB, we provision with the 2TB.

On the other hand, the second approach is to use only the value specified by the user. The reason for this approach comes from possible discounts for specific instance types, where you can not use them to the full power. An example of this is the AWS free tier. The AWS free tier is the tier two micro instance. However, to have a free access the users are limited to 30GB of disk space only, instead of the 2TB.

4.3.2.5 Workflow Orchestration

Once the cloud deployment was accomplished, we aimed to automate the execution of a chain of WPSs, a workflow. To achieve this, we use Workflow Engines. A Workflow Engine is responsible for managing workflows. The users begin by specifying a workflow with the WPSs they want to chain along the WPSs URL, and the inputs required. At the end, the users submit the workflow to the Workflow Engine for its execution. The Workflow Engine then proceeds to send requests to the WPSs in the workflow by their order. As it is possible to observe, Workflow Engines assumes the WPSs is already running. Therefore, it is required to orchestrate the trigger of the cloud deployment pipeline for each WPS before their execution. Nonetheless, it is also required to perform a dynamic validation using WPS-V after each WPS execution. As a result, it is required to use breaking points during the execution of the workflow, before and after the execution of a WPS. To illustrate this procedure we draw a state machine, Figure 4.3.

When a workflow execution is triggered, we load the workflow engine with the workflow and search for the first WPS in the chain. Once we have this information, we begin by initiating the cloud deployment pipeline. Subsequently, we notify the workflow engine to start the workflow execution. Thus, the workflow engine performs an execution request to the WPS deployed. At the end of the execution, the control is passed to our system to send a request to the WPS-V, running in the same cloud instance, to validate the outputs acquired. Since this is the first WPS in the chain, it is not possible to terminate it. The outputs obtain from its execution are going to be utilized as inputs for the next WPS. Therefore, the cloud instance must be accessible for the next WPS. Subsequently, we repeat the same process for the next WPS. It is important to mention, it is required to specify the output URL as an input during the execution request and notify the workflow engine to resume its execution. Following the execution, we do not require the previous WPS to be running, since we are not going to utilize its outputs. Thus, we terminate its cloud instance, step 7 of the state machine in Figure 4.3, and repeat the process until the workflow finishes. The last WPS is left running in order for the user to have access to the results.

Considering this, we began by concentrating on developing our own Workflow Engine in order to test the waters. Simply put, this allows us to improve the understanding of how to develop the state machine described above as well as to pinpoint with more precision the location of errors and mistakes along the process without worrying about the problems that may be caused by the Workflow Engine itself.

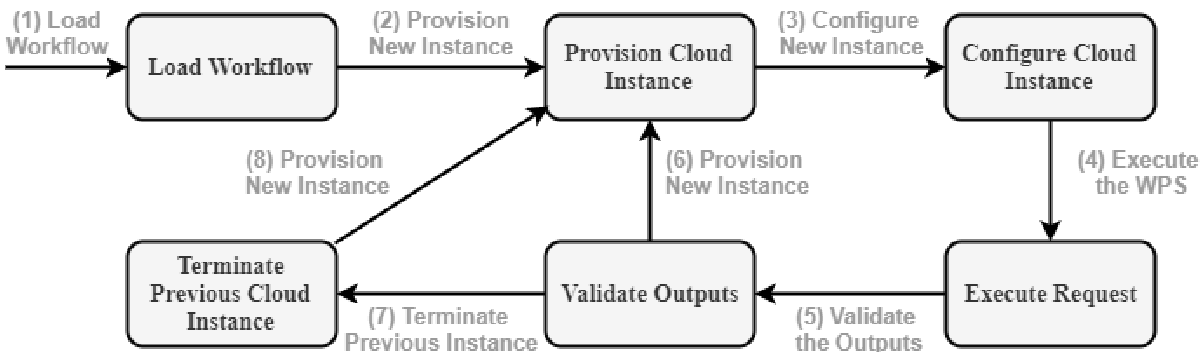
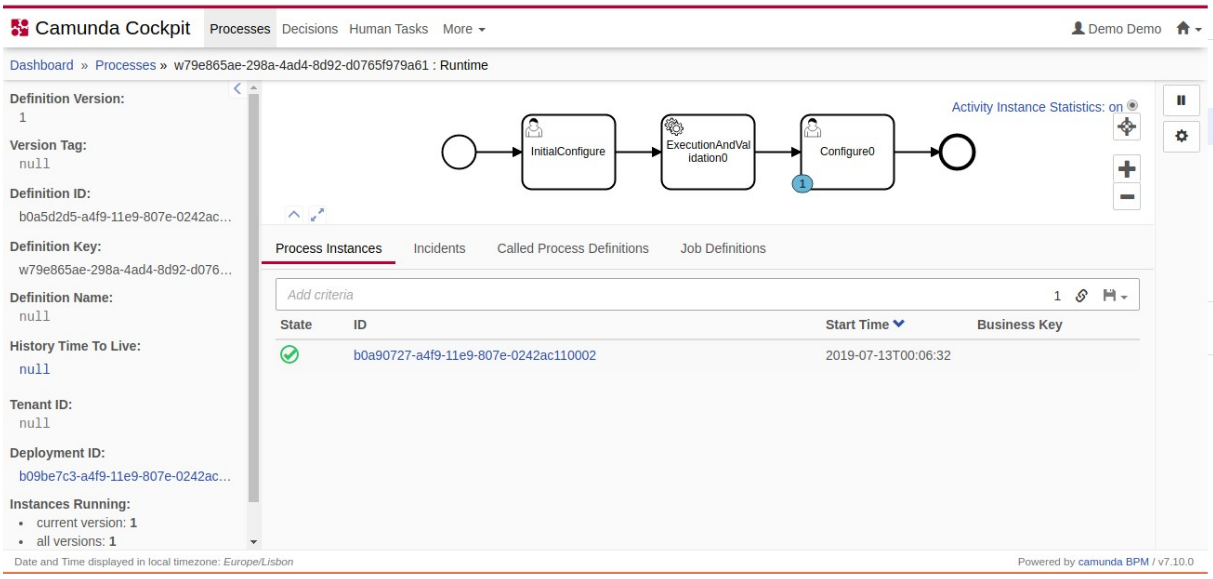

(1) Load Workflow
Load Workflow
(2) Provision New Instance
Provision Cloud Instance
(3) Configure New Instance
Configure Cloud Instance
(4) Execute the WPS
Execute Request
(5) Validate the Outputs
Validate Outputs
(6) Provision New Instance
(7) Terminate Previous Instance
Terminate Previous Cloud Instance
(8) Provision New Instance

Figure 4.3: Workflow execution state machine

4.3.2.6 Augmenting the system with more workflow engines

Down the line, when the system was fully capable of executing a workflow with the WPS running in the cloud using our workflow engine, we augmented our set of workflow engines with Camunda. Camunda is the workflow engine currently being use in the EO community. It is important to notice one of our goals was to not bind our system to any workflow engine. Workflow engines might have unique characteristics that differentiates them from each other as well as users may have preference when selecting one. While our workflow engine utilizes simple rest calls, Camunda has available a graphical user interface that enables the users to observe the execution in real time, Figure 4.4. Hereby, we provide a certain degree of flexibility by having multiple workflow engines available to the user when he desires to execute a workflow.



Dashboard » Processes » w79e865ae-298a-4ad4-8d92-d0765f979a61 : Runtime

Definition Version:
1
Version Tag:
null
Definition ID:
b0a5d2d5-a4f9-11e9-807e-0242ac...
Definition Key:
w79e865ae-298a-4ad4-8d92-d076...
Definition Name:
null
History Time To Live:
null
Tenant ID:
null
Deployment ID:
b09be7c3-a4f9-11e9-807e-0242ac...
Instances Running:
- current version: **1**
- all versions: **1**

State	ID	Start Time	Business Key
✓	b0a90727-a4f9-11e9-807e-0242ac110002	2019-07-13T00:06:32	

Date and Time displayed in local timezone: *Europe/Lisbon*
Powered by camunda BPM / v7.10.0

Figure 4.4: Camunda graphical user interface

To fully integrate Camunda in our system, it was required a week. However, the major difficulty was not in adapting it to our system but rather in learning how to utilize its API. Contrary to our workflow engine, Camunda utilizes BPMN. Since BPMN is a business process notation and JSON is an exchange data format, there is no direct relation between the two. Therefore, our first challenge was to translate the workflow specification in JSON to BPMN.

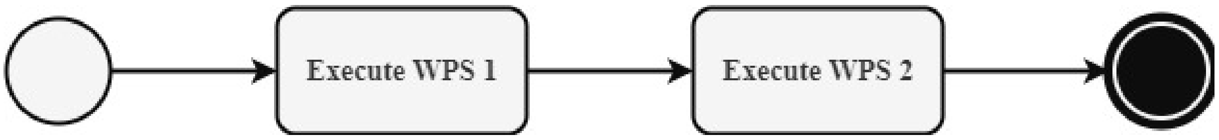


Figure 4.5: BPMN workflow diagram

Camunda has available a REST API and a Java API. Considering our system was developed using python, REST API would be the perfect solution. Howbeit, REST API does not provide means to either create a workflow dynamically, from scratch. To accomplish this, we had to integrate a Java module that uses the Camunda Java API. It is during this transformation process that we attach the breaking points as the form of External Tasks. These External Tasks enable our system to gain control and run pieces of code, the provision, configuration and validation. Through this, in lieu of modifying Camunda, we modify the workflow itself before being deliver to Camunda, easing Camunda integration in our system. As result, the workflow from Figure 4.5 is modified internally to the workflow in Figure 4.6. It is important to remember, the second WPS is left running in the cloud instance in order for the user to have access to the outputs. Most of the time was spent in this phase as the documentation to accomplish this goal was severely lacking.

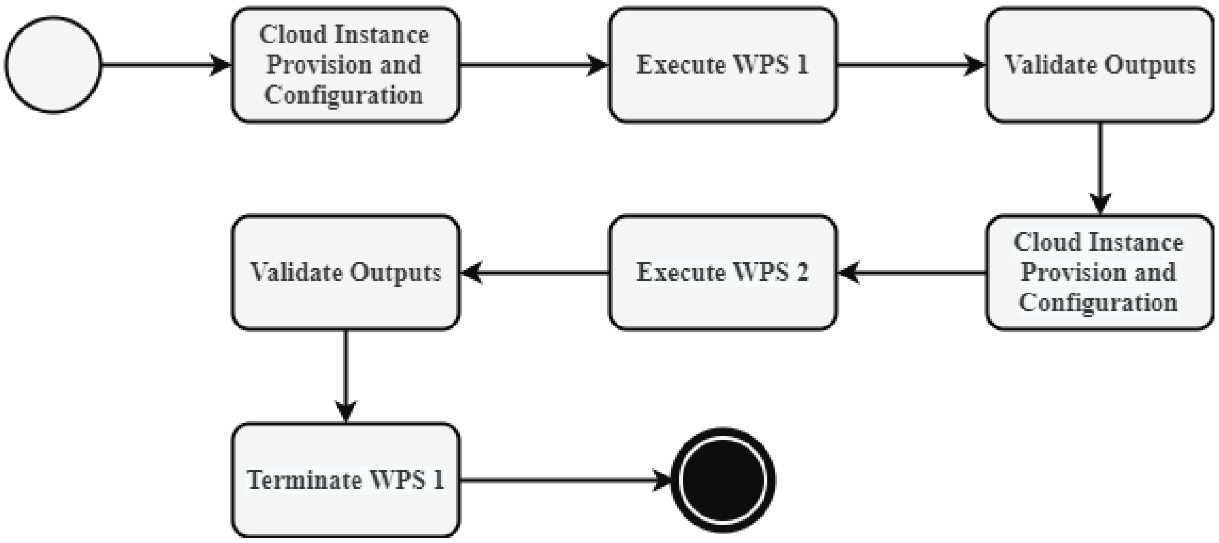

Cloud Instance Provision and Configuration
Execute WPS 1
Validate Outputs
Cloud Instance Provision and Configuration
Execute WPS 2
Validate Outputs
Terminate WPS 1

Figure 4.6: Modified BPMN workflow diagram

4.3.2.7 Execution Modes

At the end of developing the workflow orchestration, we searched for more methods to decrease the deployment process time. From this, we emerged with Execution modes. Execution modes are nothing more than different methods to perform the deployment. In other words, it is stated how to group WPS into a single instance, if they were meant to be executed in the same cloud region. By grouping WPS together, we avoid losing time preparing another cloud instance when we can reutilize the previous. However, users must be careful when stating the storage necessary, since the outputs from the WPS's executions are available on the same instance.

Our system provides three different execution modes. The first is the one we used throughout the developing of our system. We deploy a WPS per cloud instance. The second is the deploy and execute each WPS in the same cloud instance. Simply put, imagine the user wants to deploy four WPS's. For the first two WPS the region chosen was Ireland in AWS and the last two WPS were chosen to be deployed in Frankfurt in AWS. If this execution mode was chosen, we would start by grouping them by region. The first two WPSs would belong to a group and the last two WPSs to a different group. Prior to provisioning, our system will check the first group in line and verify the first region is Ireland. Upon provision and configuring the cloud instance, the system will verify which WPSs are present in the group and order the Docker in the cloud instance to pull and deploy these WPSs. Once the execution of the first WPS is completed, our system would verify the following WPS is in the same group and would skip the deployment pipeline. Once the second WPS finishes the execution, we proceed to repeat the same procedure. Through this we can save time. However, there is a catch in this execution mode. It is important to mention, WPSs have different hardware requirements. Through this mode, we only group WPSs if they have the same hardware requirement. In other words, if one of the WPSs of Ireland requires four gigabytes of RAM and the other WPS requires twelve gigabytes of RAM, they would not be grouped together. To accommodate this case, we developed a third execution mode. Deploy and execute each application in the same cloud instance using the highest WPS hardware requirements. In the previous example, the cloud instance of the first group would have the twelve gigabytes of RAM.

To augment our system with these execution modes, it was not required any major modifications. As observed above, our system performs a filter to group the WPSs together before triggering the deployment process pipeline. Afterwards, to verify if it is required a new instance it simply checks the groups. With the Ansible playbooks were categorized by provision for each cloud provider, configuration and Docker deployment, our system only had to use the Docker deployment playbook for each WPS required. The only mandatory change was to the Nginx configuration template. In this, instead of just specifying one WPS, it may be required to specify more now. To enable this, we appended a for loop to it. Considering the use of a tag to specify the location to the WPS in the URL, we are able to avoid URL conflicts between the WPSs. As an illustration, we have a WPS to perform aggregations, other to calculate LTA and 196.1.2.3 as a machine ip. To access the aggregations WPS we would use the following URL 196.1.2.3/aggergations/. On the other hand, to access the LTA WPS we would use the following URL 196.1.2.3/lta/.

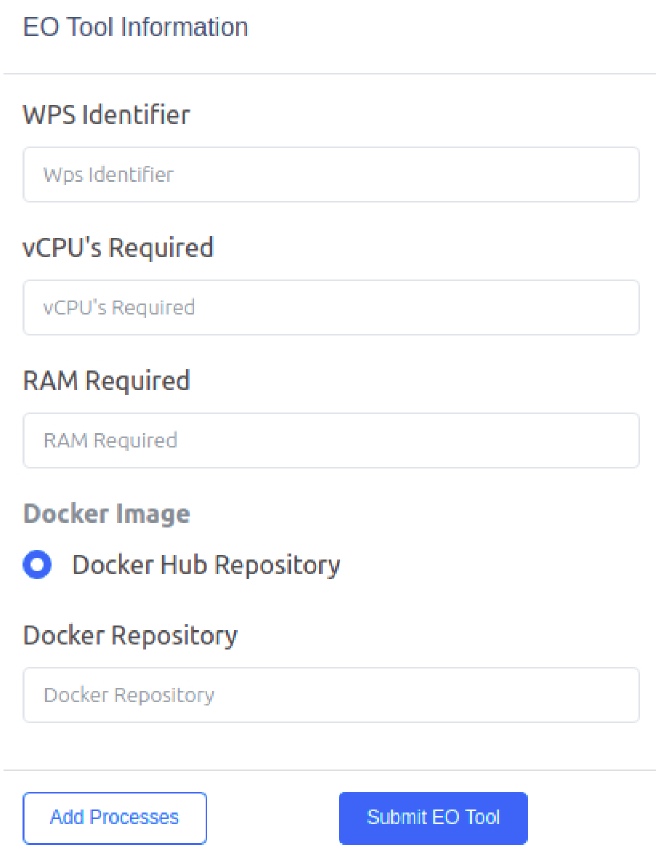
4.4 MCDEEOS-WEB

To enable the users to interact with the system, we design a clean and simple interface available through our front-end module, MCDEEOS-WEB. As previous mentioned, to access our system users are required to register or login first. The information requested is of two types. The first corresponds to personal information, the username, password and email. The second is the cloud keys. These are used to grant our system access to enable deploys in the respective cloud providers. Afterwards, the users have full access to our system. They are able to package applications, create and execute workflows into the cloud and observe all the available WPS in the system. At the end of developing our system, new requirements appeared. The OGC, along with a few companies, offers information regarding WPSs already running in a server through the use of catalogs. We enhanced our system with WPSs discovery from these catalogs along with additional features. The registration and removal in a specific catalog of a WPS already running in a server as well as the search for specific WPSs from a specific catalog.

4.4.1 EO tool Packaging

The EO Tool Packaging enables the transformation of an EO tool into a WPS. To accomplish this, users are required to provide information regarding the EO tool. This information is compelled by two categories. As seen in the Figure 4.7, the first category is related to the EO tool itself. One key component of our system is the ability to share EO tools between the users of the system. A user that wants to utilize a WPS from another user, may not know the necessary hardware requirements of it. To avoid choosing the incorrect configuration and as a result wasting resources, this information is specified during the EO tool packaging. We consider the user wanting to package the application to be the one with the most knowledge about it, i.e, being its developer. Along with the specification of the RAM and CPU required, the location of the EO tool must be specified along with the desired name. The application location must be a docker repository, since we will use it as base for our wrapping process.

The second category is the information regarding the EO tool processes. An example of this, a EO tool image processing might have processes for cropping, resizing or create an aggregation of several images. All EO tools must have at least one process. These processes can have different inputs and outputs. The OGC defined a standard in which the data types are grouped into three categories, LiteralData, ComplexData and BoundingBoxData. Taking all this in consideration, WEFEOS-WEB provide a web-form for the user to describe all this information, as observed in the Figure 4.8.



EO Tool Information

WPS Identifier

Wps Identifier

vCPU's Required

vCPU's Required

RAM Required

RAM Required

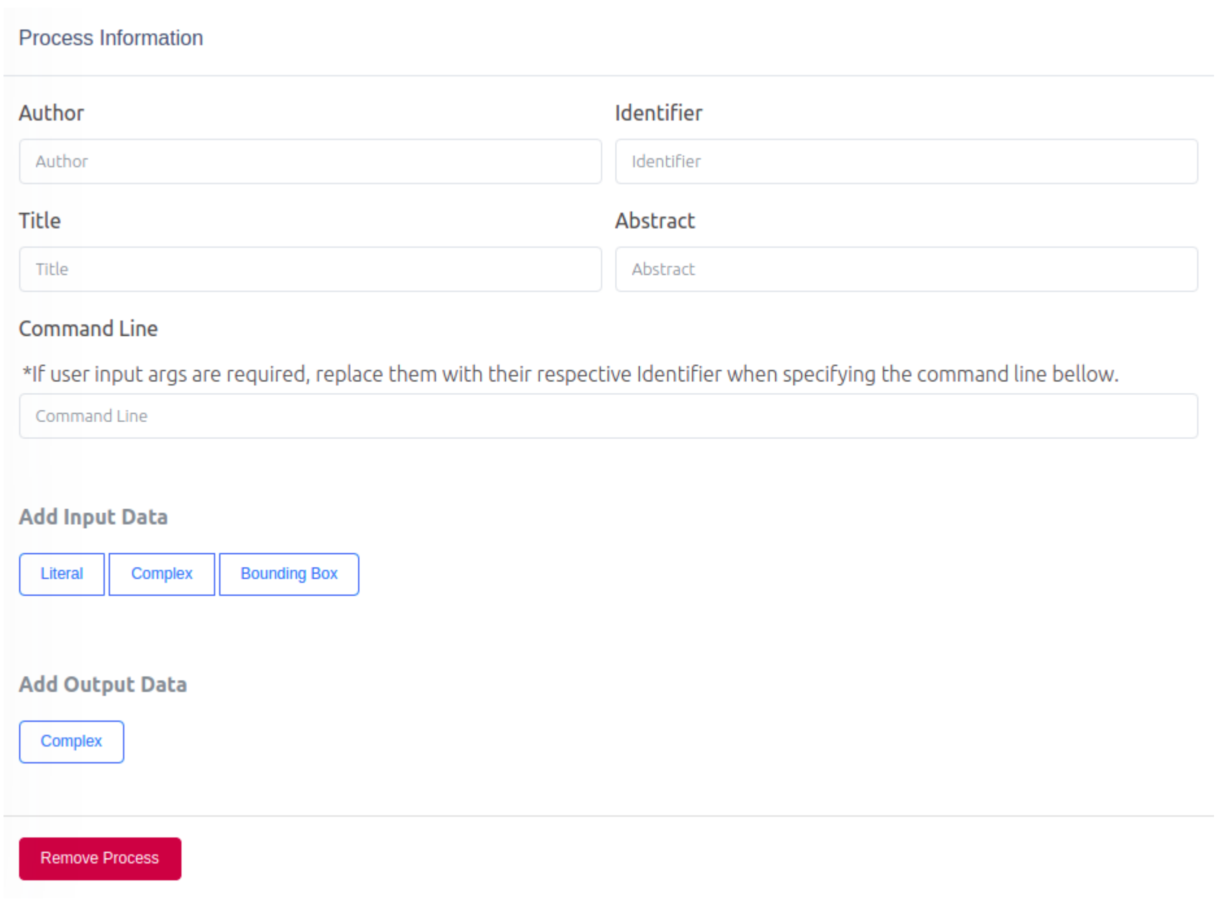
Docker Image

[✓ Docker Hub Repository]

Docker Repository

Docker Repository

Figure 4.7: Information required about the EO tool for its packaging


Process Information

Author
Author

Identifier
Identifier

Title
Title

Abstract
Abstract

Command Line

*If user input args are required, replace them with their respective Identifier when specifying the command line bellow.

Command Line

Add Input Data

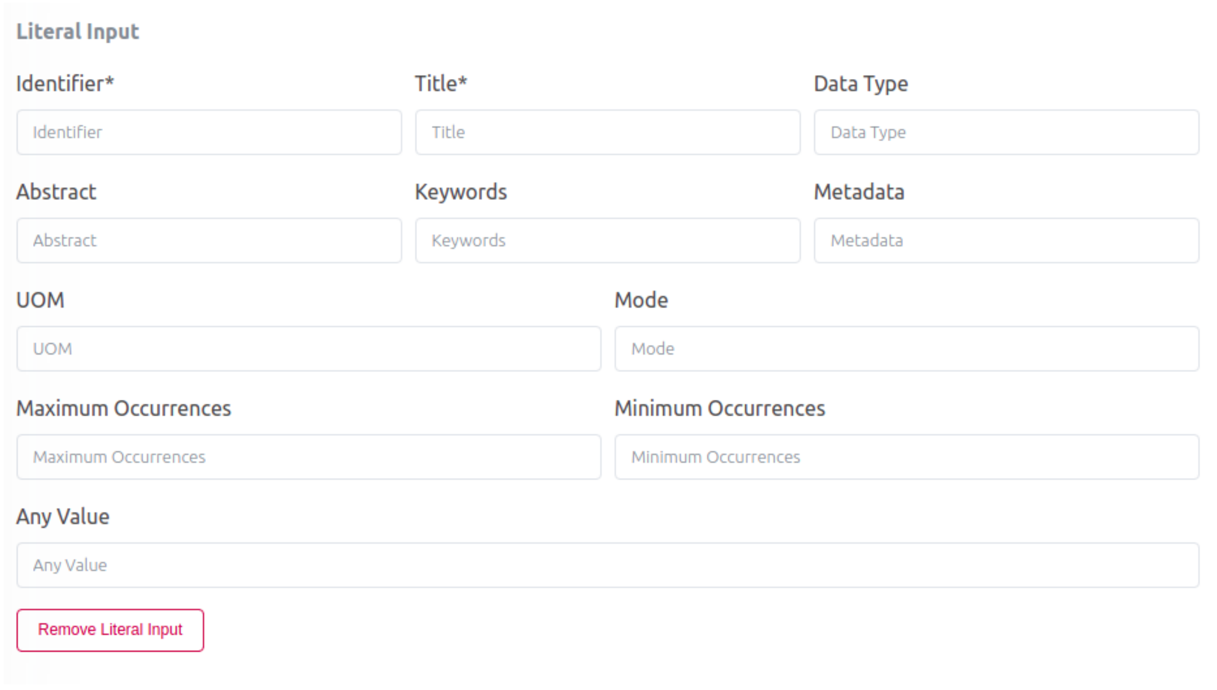
Literal | Complex | Bounding Box

Add Output Data

Complex

Remove Process

Figure 4.8: Information required about the EO tool processes for its packaging


Literal Input

Identifier*
Identifier

Title*
Title

Data Type
Data Type

Abstract
Abstract

Keywords
Keywords

Metadata
Metadata

UOM
UOM

Mode
Mode

Maximum Occurrences
Maximum Occurrences

Minimum Occurrences
Minimum Occurrences

Any Value
Any Value

Remove Literal Input

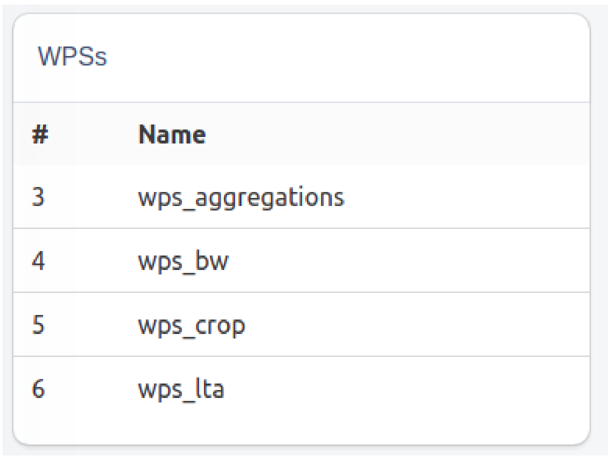
Figure 4.9: Information required about the ComplexOutputData for the EO Tool Packaging

In the event of users wanting to specify more inputs or outputs, we provide buttons to append more web-forms for this. It is important to mention, even though the outputs have three possible data types, only ComplexData can be worked with. With the possibility of a WPS generating multiple LiteralOutputData, it is not feasible to detect and parse each LiteralOutputData. Another major reason is through observing the WPSs available in the industry they all work through files, therefore only ComplexOutputData is generated. Along with the data types users must also specify the location where the outputs will be stored inside the docker containers, Figure 4.9. This allows our system to make them available in the future for the chaining process. Nevertheless, we also provide fields for metadata as the author, title, description, abstract and the most important the command line needed to execute the EO tool. The full path must be specified for the EO tool in this command line. As previously mention, a WPS might have several processes, the Add Process button present at Figure 4.7, enables the addition of a new web-form for the specification of the other processes.

4.4.2 List WPSs

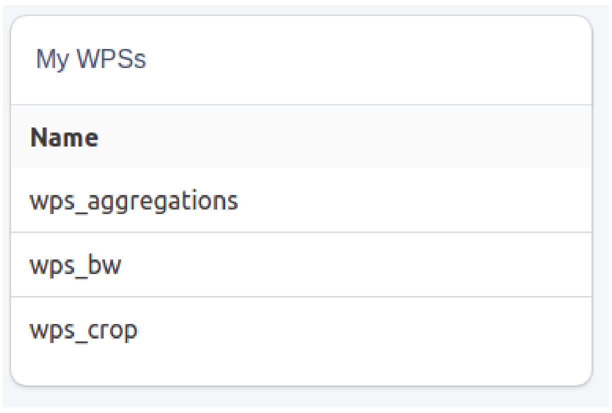
With several EO tools being packed into a WPS using our system and, therefore, registering in it, MCDEEOS-WEB is able to present all available WPS to the users.

Through the use of a table, Figure 4.10, it is displayed every WPS the users owns. As stated above, one key component of our system is the ability to share WPS between users. For this reason, it crucial to also display the WPSs that belong to other users. In other words, all WPSs present in the system. To accomplish this, we provide another table identical with the users own WPSs table, Figure 4.11.


WPSs

#	Name
3	wps_aggregations
4	wps_bw
5	wps_crop
6	wps_lta

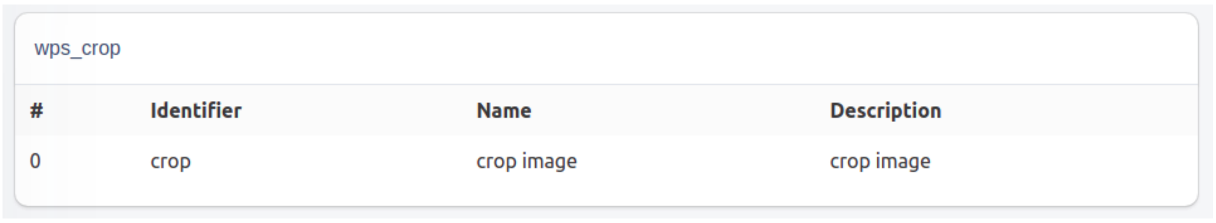
Figure 4.10: List of WPSs present in the system


My WPSs

Name
wps_aggregations
wps_bw
wps_crop

Figure 4.11: List of the WPSs belonging to a user

In case a user required more information regarding a WPS, the name of the WPS in the table must be clicked, triggering an event to create a new table with the list of processes available in the WPS along with their identifiers, titles and description, Figure 4.12.


wps_crop

#	Identifier	Name	Description
0	crop	crop image	crop image

Figure 4.12: Description of a WPS name wps_crop

4.4.3 Workflow Execution

To develop MCDEEOS-WEB we decided to use the same programming language as WPS-V-WEB, Vue-js. The objective for this was to ease the integration of the WPS-V-WEB module. Vue-js enables to import different projects as components into other projects. However, due to the different version on the Vue-js used in WPS-V-WEB some modifications were required to turn this into a component. In the end, the fully integration was accomplished without any feature loss or malfunction in the Workflow Execution section of the MCDEEOS-WEB. Workflow Execution section can be seen in two different parts, the first creation and validation of the workflow and the second the specification of cloud information.

With the success in use WPS-V-WEB, its Graphical User Interface (GUI) remained the same. It has available a drag and drop dashboard with a menu for creating WPS, start-event, end-event, importing workflows and save workflows, Figure 4.13. When selecting a new WPS a new web-form is available with dropdown boxes to select which WPS and process the users wants along with a name for the box representing the WPS.

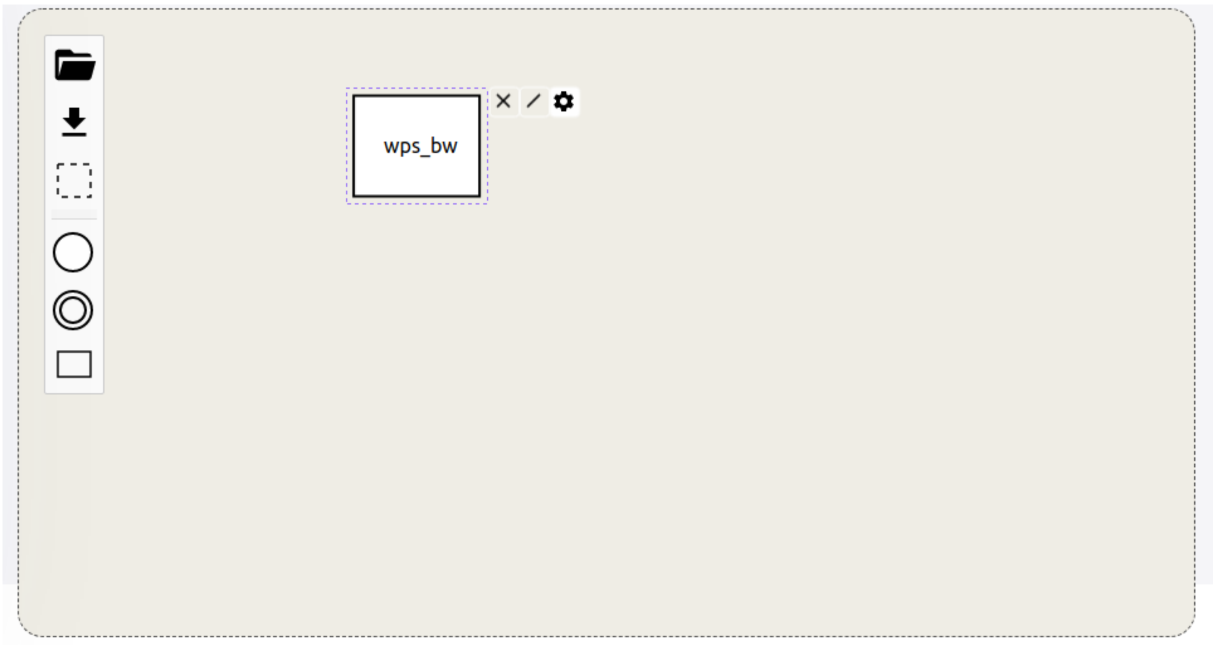
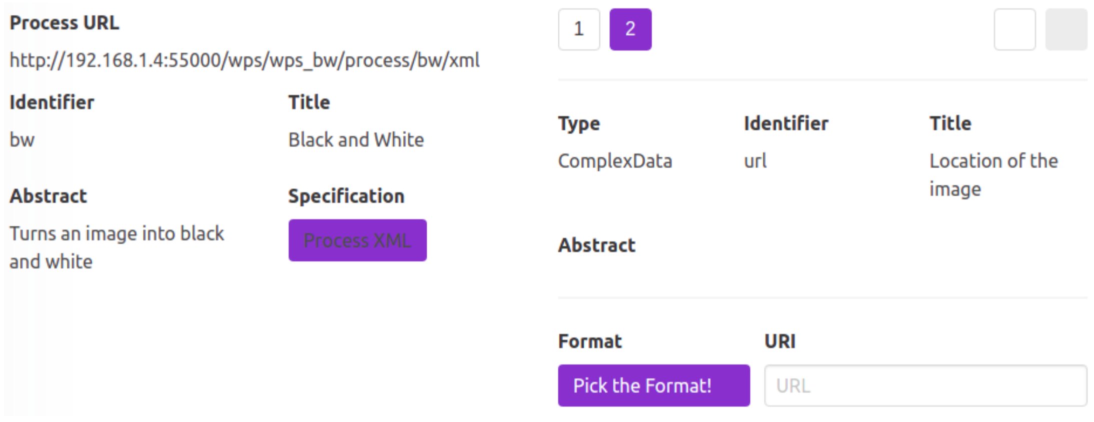


Figure 4.13: Dashboard for the workflow creation

With enough WPS, users are now able to chain then to create an actual workflow. At the end of chaining, users must specify every information required for the inputs of each WPS and its chaining, as seen in the Figure 4.14.



Process URL
http://192.168.1.4:55000/wps/wps_bw/process/bw/xml

Identifier
bw

Title
Black and White

Abstract
Turns an image into black and white

Specification

Type
ComplexData

Identifier
url

Title
Location of the image

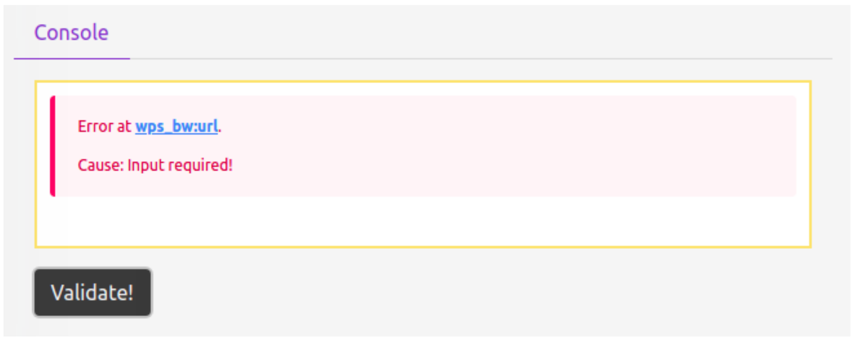
Abstract

Format

URI

Figure 4.14: WPS process specification

Finally, with every important detail is specified, users are able to validate. WPS-V-WEB has available a console, Figure 4.15, in which error messages are displayed resulted from the validation, to help the user correct the workflow.

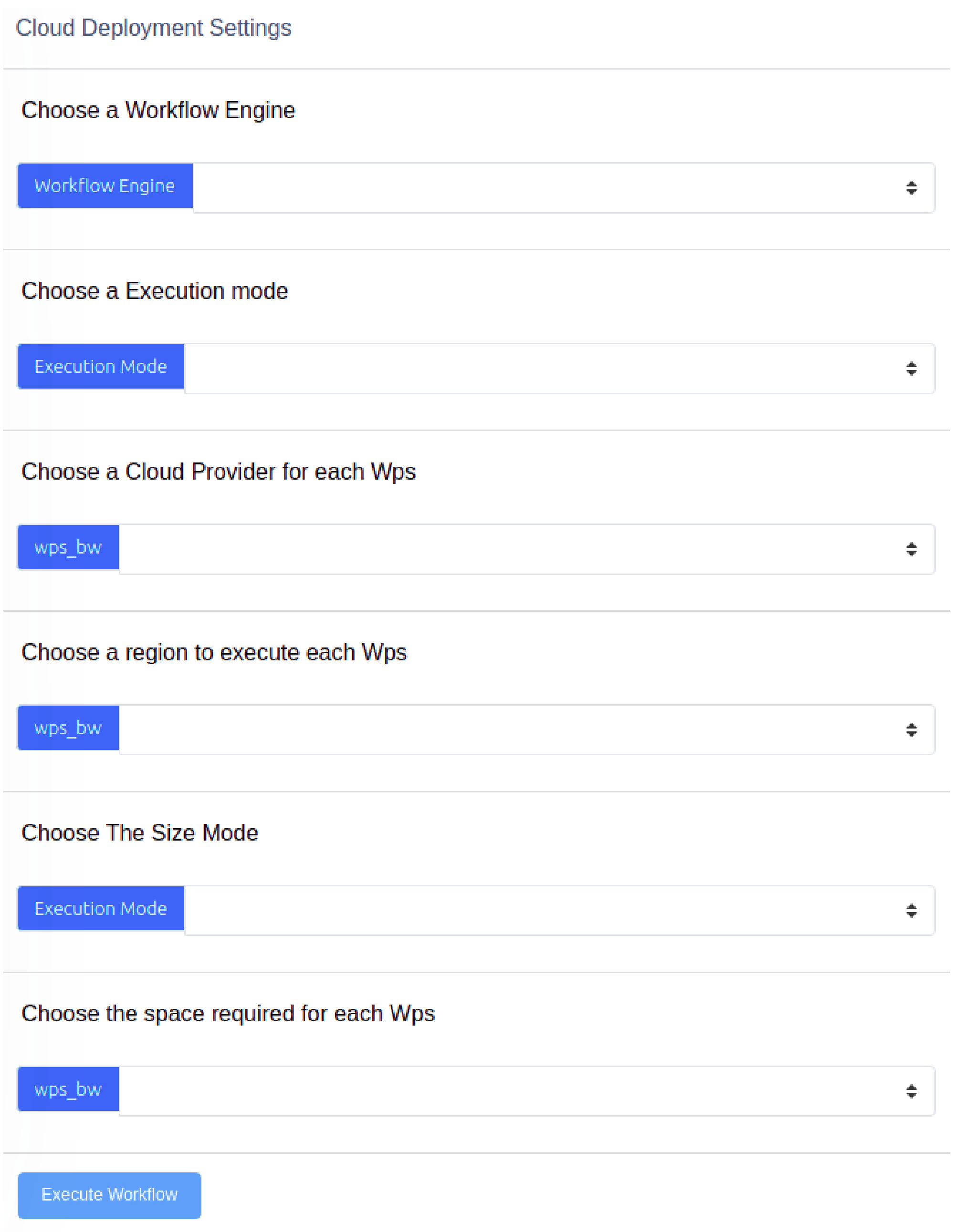


Error at wps_bw:url.

Cause: Input required!

Figure 4.15: Console displaying the errors detected during the workflow validation

Once the workflow is successfully validated, the button to execute the workflow is enabled. However, users still need to provide critical information. To execute a workflow, we first need to load it into a workflow engine for its execution. When developing our system, we observed that in this industry the concept of workflow is recent. As a result, different workflow engines are arriving the market. We did not wish to bind our system to a specific workflow engine, but instead be flexible enough to increase the available set and let, through the dropdown box, the users choose the one that fits best their needs. We also provide different execution modes through a dropdown box, Figure B.14. Subsequently, the users simply have to select the cloud provider available and the region for each WPS specified in the workflow. At the end, it is also required to specify the size mode for each instance along with the necessary size. With everything specified, the users are ready to execute the workflow in the cloud.



Cloud Deployment Settings

Choose a Workflow Engine

Choose a Execution mode

Choose a Cloud Provider for each Wps

Choose a region to execute each Wps

Choose The Size Mode

Choose the space required for each Wps

Figure 4.16: Cloud information required for the workflow execution

4.4.4 Register WPSs

The interface, Figure 4.17, to enable the registration of services into the different catalogs is relatively straightforward. In order for the users to accomplish the registration, first, a catalog must be selected for the service registration in the dropdown menu. Subsequently, the users only require uploading a Geojson with every crucial detail of the service for the registration.

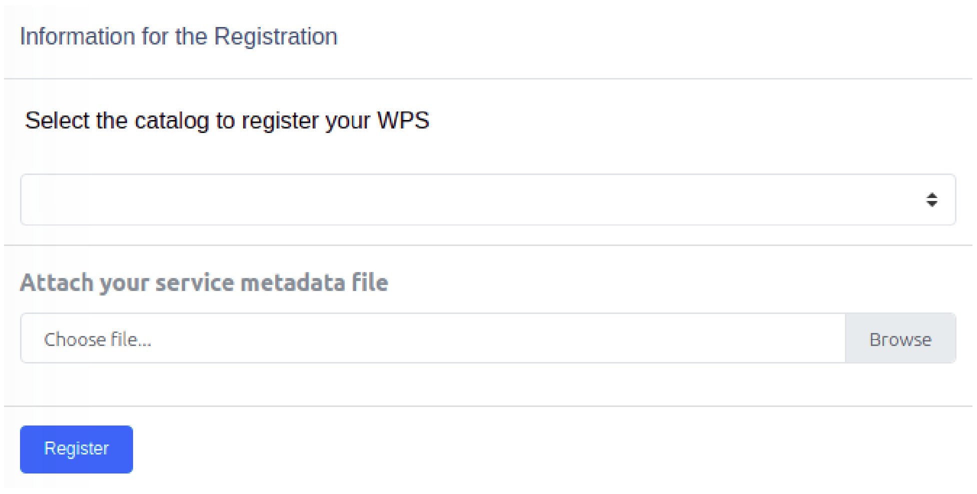

Information for the Registration
Select the catalog to register your WPS
Attach your service metadata file
Choose file...
Browse
Register

Figure 4.17: Interface for the WPS catalog registration

4.4.5 Remove WPSs

To remove a service from a catalog, it is presented two dropdown menus, Figure 4.18. The first is used to specify the catalog in which the service is present. The second is to specify the service identifier. It is important to mention users can only remove services they registered. As a result, our dropdown menu presenting the services only contains the services belonging to the user.

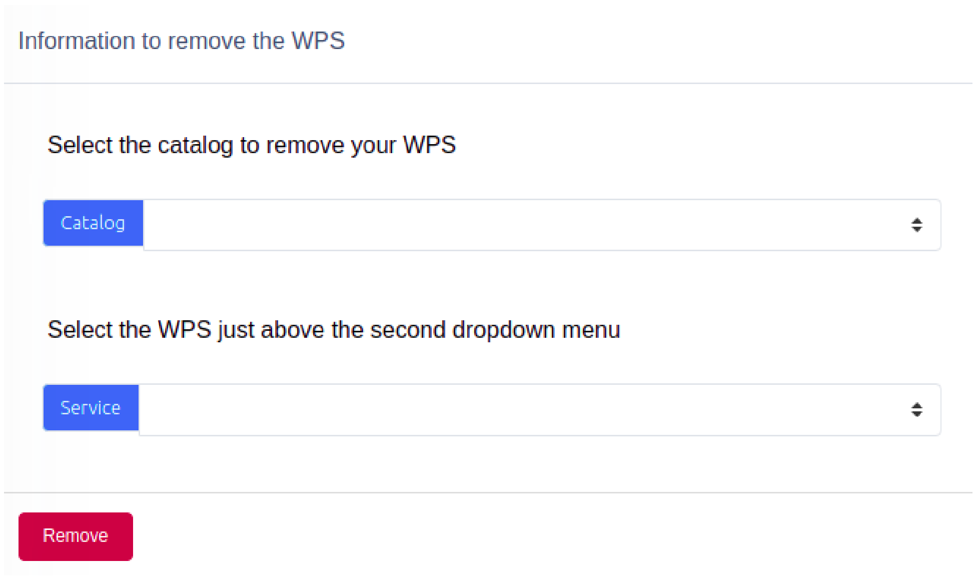

Information to remove the WPS
Select the catalog to remove your WPS
Catalog
Select the WPS just above the second dropdown menu
Service
Remove

Figure 4.18: Interface for the WPS catalog removal

4.4.6 Discover WPSs

In case a user wants to search for a service, we provide a service discovery. In other words, the user provides a part of a identifier name, for example Deimos, and our system will present, through the use of the same tables as in subsection 4.4.2, every service if the identifier contains Deimos. Along with every service, we also provide every process belonging to the service. Figure 4.19 displays the interface to accomplish the discovery. A dropdown menu for the user to select the catalog, a textbox to allow the user to specify the name for the search and the tables for the Deimos example.

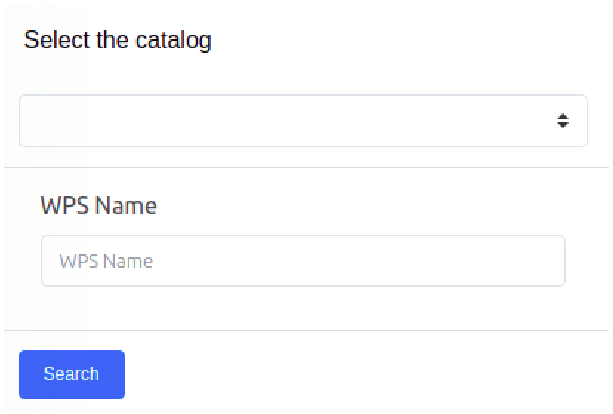
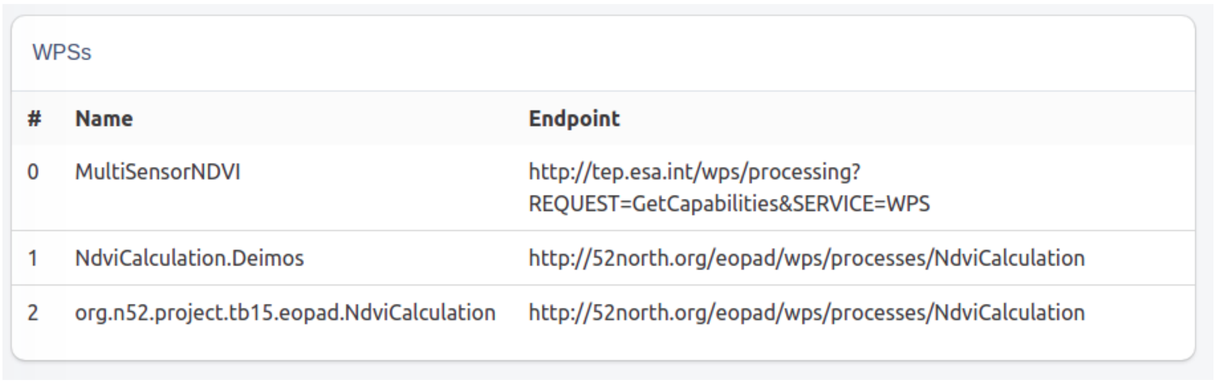


Figure 4.19: Interface for the WPS catalog discovery

4.4.7 List WPSs from Catalogs

In the same vein as the subsection 4.4.2, our system also displays the WPS present in every catalog through the use of tables. In spite of presenting every WPS available in every catalogue, to facilitate the navigation, our system provides a dropdown menu for the users to select the catalogue they want to see. Since every WPS is already running in a specific server, our tables have one more column to mention the Endpoint URL, Figure 4.20. Through this Endpoint URL it is possible to contact the WPS. Equally to the subsection 4.4.2, we are able to click upon a WPS to obtain more information and have an identical table with the WPS belonging to the user.



WPSs

#	Name	Endpoint
0	MultiSensorNDVI	http://tep.esa.int/wps/processing?REQUEST=GetCapabilities&SERVICE=WPS
1	NdviCalculation.Deimos	http://52north.org/eopad/wps/processes/NdviCalculation
2	org.n52.project.tb15.eopad.NdviCalculation	http://52north.org/eopad/wps/processes/NdviCalculation

Figure 4.20: List of WPSs present in a catalog in the system

Chapter 5

Validation

In this chapter we aim to validate our solution. To aid in this evaluation we will first revise the set goals. Subsequent, the validation will be done in two parts corresponding to the EO Tool Packaging and Workflow Execution in the cloud environment. In each of these, we are going to describe the possible approaches to accomplish them and which we consider the best to help us evaluate our system. Moreover, we provide the description of the steps required for the accomplishment of the EO Tool Packaging and Workflow Execution in the cloud environment, list of possible errors to keep track, how the execution of the validation will take place, the results obtained along with an analysis of these.

5.1 Goals

To perform a thorough validating for our system, it is first required to define means of comparison between the two approaches. In other words, we set goals based on the desired functionalities by the EO community. As mention above, we want to provide cloud computing as a solution while providing the finest user experience, without placing extra burden on the user. As a result, we can translate this into the following usability requirements:

- The end users learn with ease how to use the system;
- The end users use with ease the system;
- The system is able to prevent and recover from errors.

It is important to have in mind the only approach to cloud computing solution currently is the manual approach. As a result, the manual approach will serve as the base of comparison to our solution.

5.2 EO tool Packaging

The validation of our system is done in two parts. The first part corresponds to the EO Tool Packaging. This is composed by three phases, EO tool wrapping into a WPS, WPS encapsulation into a Docker image and image storage.

It is important to mention, the procedure of wrapping an EO tool generally has an impact in the performance of the application when not performed case by case. Traditionally the addition of a new layer may involve transformations for the inputs as well as modifications to adapt the code. However, in our situation the new layer aims to provide descriptive information regarding the EO tool itself through HTTP. An illustration of this, is the delivery of an XML file containing information about who created the EO tool, what the EO tool does, which types of inputs and outputs it requires. For example the SayHello process XML file present at Listing A.1 in Appendix B. In addition, it also provides a method to execute the process. In other words, we are able to execute and have access to the results of the EO tool through a POST request. The inputs do not require any transformations, since they are passed to the program as sent via the post request. The EO tool itself does not require any modification as well. The new layer simple calls the EO tool with the inputs sent, as explained in the previous chapter. As a result, performing the wrapping manually or automated is the same. Automation of the wrapping will not result in a decrease of performance. It is important to mention, even though it does not interfere with the performance itself, by performing the EO Tool Packaging through our system, errors resulting from the EO tool are not recorded. It detects the presence of an error but not the full details of line where it occurred and its type. While doing the EO Tool Packing through the manual approach it is possible to record the errors with these details.

With our system, all these phases are automated. However, the users ought to provide information regarding the EO tool. This information is categorized into three sections. The first is about the EO tool itself. The users specify the name, the RAM and CPU required along with the Docker hub repository. The second section is about the process metadata. This includes the name of the author, the title, abstract and the identifier of the process. Finally, the most crucial information required is the command line to run the EO tool and the definition of the inputs and outputs metadata. As can be seen in the chapter 4 subsection 4.4.1, one possible limitation of using our system to provide automation is the specification of the outputs in terms of system expressiveness. Doing the process manually would enable to user to specify the different LiteralData and BoundingBoxData obtained from the execution of the EO tools. We cannot achieve this through automation since these results would be written to the console and it is not feasible to detect and parse these data types. However, in the EO field the EO tools only produces ComplexData types. In other words, the EO tools only generate files as the output. Thus, we do not loose expressivity when using our system, since the only output data type utilized is supported by our system, the ComplexData. At the end of filling all this information, our system is ready to perform the automatization of the EO Tool Packaging. Doing the process manually is different than using our system in the eyes of the users. As a result, the validation of our system is measured through the filling of the information.

5.2.1 Manual Steps

When performed manually, there are several possible approaches. These differ from method utilized to technology. As we go through the EO tool package phases, we will mention the different possibilities.

Therefore, the first phase required is the wrapping of the EO tool into a WPS. A WPS is a web service characterized by providing three standard operations, through HTTP Get and Post methods. These operations are GetCapabilities, DescribeProcess and Execute. The process of wrapping an EO tool into a WPS consists in developing these REST methods in the EO tool. To perform these, users have two possible approaches. Develop from scratch the REST endpoints or utilize as base an already existing implementation of a WPS and adapt to the EO tool, i.e PyWPS. In the context of validation, we will adapt the EO tool to the existing implementation of a WPS, since creating the endpoints from scratch would require more time. At the end of wrapping the EO tool into a WPS, it is required to package it together with all its dependencies. As described in chapter 3, there are several technologies available, each with their own limitations. Therefore, we used Docker since it is the technology with less complexity as well as more flexible. Docker allows us to create an isolated and portable environment, tailored specifically to the application. To perform the encapsulation, the users write a Docker file with at least five commands. First, it is required to select as base an image that provides the adequate operating system. From this the users must update and install all the dependencies, create a directory to copy the WPS into it and provide the command to run the EO tool. Similar to the Listing 4.4 in subsubsection 4.3.1.1 . Once the Docker file is written, the users are able to build a Docker. It is important to mention, users are free to skip this phase. However, they need to transfer the WPS directory to the cloud instance using the SSH connection and installing the dependencies each time a new cloud instance is deployed. As a result, this method may be considered faster early on, but later during the cloud deployment, it will be worst since it always requires installing all the dependencies and, therefore, will not be consider for the validation.

At last, the newly encapsulated WPS should be stored somewhere. In the same vain as before, the users are able to pass the newly encapsulated WPS with all its dependencies through the ssh channel. In our case, by using Docker to perform this, the image must be first compressed into a ZIP file, only then it can be transferred. On the other hand, Docker itself provides a repository specifically for Docker images easing the process to store them as well as to operate them.

In the end, the simpler procedure, described in the state machine in Figure 5.1, can be translated into the following:

- Convert the EO tool into a WPS using PyWPS;
- Create a Dockerfile specifying each command to encapsulate the WPS and their dependencies;
- Build the Docker image through the Docker file;
- Push the Docker image to the Dockerhub.

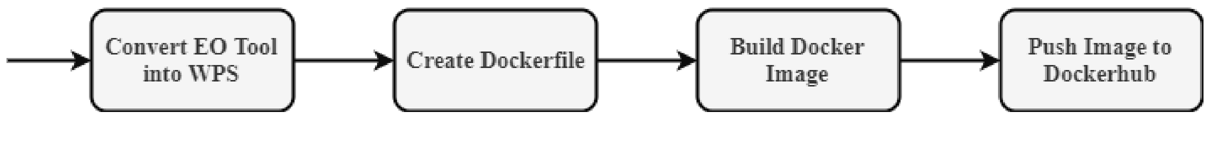


Figure 5.1: State machine describing the manual procedure for the EO Tool Packaging

5.2.2 System Steps

As mention before, our system automated the steps done manually. However, the user still needs to encapsulate the EO tool, as it is required the repository where the Docker image is present and provide additional information to ensure the correct functionality of the EO Tool Packaging. The information provided by the user is done through a web-form. This can be translated into the following steps:

- Specify the name and hardware requirements for the application;
- Specify the Docker repository for the base image containing the EO tool;
- Specify the input and output formats;
- Specify the command line to execute the EO tool;
- Specify the location of the outputs.
- Select the submit button.

5.2.3 Errors

Users are always prone to errors when performing the packaging of the EO tool in either procedure. The automation of the EO Tool Packaging requires user input to be accomplished successful. However, errors may still occur in the event of the users providing either wrong information or malformed images. Considering that our system is not able to pinpoint which error of the previous errors took place, it is still possible for the users to verify the information provided since it is not erased from the input fields. This allows to further restrict the possibilities of where the error happen and also to free the user from filling all the information again from scratch. Upon correcting the errors, the users must submit again to start the process from scratch. The potential errors when using our system can be reduced to the following list:

- Misinformation during the specification of the inputs and outputs data;
- Missing information during the specification of the inputs and outputs data;
- Wrong Docker repository;
- Malformed base application Docker image provided.

Regarding the manual approach, the errors made by the users can be traced more precisely since the errors are commonly related to the step of the procedure in which the users are. In addition, the users do not require to restart the whole process of encapsulation if a mistake happen, the work previously done is stored and just need the correction of that error. Despite these advantages, the possibility of errors by doing the process manually is far greater than automated as well as these errors may be more complex. The following list describe the possible errors when doing the packaging manually, where we highlight the possible mistakes in both procedures:

- Mismatch between functions definition and PyWPS version;
- **Misinformation during the specification of the inputs and outputs data;**
- **Missing information during the specification of the inputs and outputs data;**
- Coding mistakes when translating the EO tool to WPS;
- Missing the initialization of the new process class in the PyWPS code;
- Malformed Dockerfile. For instance, missing or wrongly specified parameters, dependencies and commands;
- Failing pushing the image to the Dockerhub.

5.2.4 Execution

With the clarification on the steps required for both approaches along with the list of possible errors, we are now going to unfold how the execution of the validation of the EO Tool Packaging is going to be accomplished.

In the first place it is important to have in mind scientists are inexperienced EO developers. Their academic background is from the different fields of engineering, e.g., chemistry, physics. As a result, even though they never used cloud computing, they already possess some basics concepts and terms. In other words, they have a bare minimum knowledge in cloud computing. Their main focus is on the analysis of the results obtained through the use of EO tools, rather than developing them. Given that, our target audience, when developing our system, is the scientific community. Therefore, our validation is performed with people who fits this criterion. Each user is going to utilize their own EO tool. As it is possible to notice, the manual approach requires days to not only learn but also perform the EO Tool Packaging successfully. The complexity of adapting or develop a WPS from scratch, even with the help of the documentation, is far greater than using our system, which only requires simple information. Due to the complexity levels, it is meaningful to measure manual times, since they are far too high. Therefore, the validation is done in one phase, the automated, with the several steps described above. Prior to start, it is required to learn how to perform each step with the involving technology. Each user is going to search for information regarding the use of our system. This information is available through a guide. We measure the time spent in the whole process of learning. However, it is normal to still search information during the execution of the task. We are going to have into account this time and add it in the end. At the end of measuring the learning time, we are going to analyze the results and see if we achieved our goal of providing a system easier to learn.

At the same time, we also want to examine if our system is faster to use. We will measure the time spent by each user using our system to perform the EO Tool Packaging. It is vital to perceive we are concerned with usability and not performance. Thus, during the automation phase, the time the system requires to perform the pipeline is not going to be taken into account.

During the execution of the task, it is common for the user to make mistakes. Through the list of possible errors defined above, we will track how many the user made during the execution of each step. This allows us to validate if our system is capable to diminish and therefore prevent errors, compared to when the process is done manually.

5.2.5 Validation Script

With the goals and the tasks defined as well as how to execute them, we wrote a script for the validation with the user. We start with a brief introduction and explaining how the validation is going to proceed. A simple example of this introduction is the following text:

Hey, thank you for participating in the validation of our system. Before we begin, I am going to discuss how this validation is going to work. We developed a system that enables the automation of the packaging and deployment of an EO tool. As a result, we want to verify if our goals for the development were met. I'll be asking you to complete a specific set of tasks, while observing and recording the time in each task and step you perform along with the errors you might encounter. For each task we will be giving context on why it is necessary to be done. We would like you to tell us your thought process while doing the task if possible. Before proceeding to the validation do you have any question?

Upon answering all the questions, the user might have, we start the validation by specifying each task one at a time. The sequence of the tasks begins with the manual approach and then the system, first for the EO Tool Packaging. To be more precise we defined the following task list:

- Briefly explain the task consists in transformation of the user EO tool into a WPS using our system;
- Provide the documentation for the EO Tool Packaging of our system and ask the user to spend some time reading it. At the end, state to the user can consult it during the tasks;
- Proceed to request the user to use our system EO Tool Packaging feature.

5.2.6 Results

The validation of our system was accomplished with three users, during the third week of October 2019. The users age ranged from 32 to 43. All of the three users have a Computer Science degree, two of them are working at the moment as Technical Manager, while the other user is working as a Project Manager. Although, the users have a background in Computer Science, none of the users had used cloud computing before, only knowing the basic terminology. In addition, thee users never had interacted with our system prior to the validation, they were inexperienced users.

It is important to mention two of the usability tests were done in an isolated room, only with a computer containing the system. Regarding the third usability test, due to the lack of empty rooms available, it was accomplished in the normal environment where the user works. Even though there were more people in the room at the same time, no one interfered with the user during the usability tests.

Upon finishing the usability tests, both tasks of learning and executing were accomplished with success. We condensed the results obtained into two tables. The Table 5.1 contains the time measured for each user in seconds, regarding the learning and the execution phase. While the Table 5.2 presents all the mistakes done during the execution of the EO Tool Packaging task.

Learning Phase	Execution Phase
471	445
432	451
352	542

Table 5.1: EO Tool Packaging times recorded from the system validation

Mistakes	Number of occurrences
Misinformation during the specification of the inputs and outputs data	0
Missing information during the specification of the inputs and outputs data	1
Wrong Docker repository	0
Malformed base application Docker image provided	0

Table 5.2: EO Tool Packaging errors recorded from the validation

5.2.7 Analysis

As can been observe, our system learning curve is very low regarding the EO Tool Packaging. It averages 418 seconds to learn its usage. The reason behind the low average time, when compared to the manual process that might take a whole day to fully understand, is due to our system only requiring simple information about the EO tool, as described in the subsection 4.4.1. Therefore, our system, for the EO Tool Packaging simply describes the details required in each field, containing a more detailed description on the more complex fields. In addition, it also uses figures from the system itself to help in the visualization. In the end, it results in a simple and easy to read guide.

Furthermore, as mentioned above, the use of straightforward information with the support of the easy reading documentation translated in faster times during the execution phase. As a result, it averages 479 seconds to use the system for the EO Tool Packaging. It is important to mention during the execution phase a user forgot to fill a crucial field regarding the ComplexDataOutput. However, our system was able to detect and notify the user straight away, avoiding performing the encapsulation straight away and waste time. The EO Tool Packaging done manually required days to complete. Adapting or develop this procedure from scratch is always a burden requiring the full attention of the users and prone to errors consuming additional time to discover and fix them. In addition, the documentation would require multiple readings during the procedure due to its complexity. It is important to have in mind, the complexity of the EO tool has an impact in both time and errors. The more complex the more prone and time consuming the procedure is. However, using our system would always be safer and faster than doing the procedure manually, since its complexity is always bigger than using our system.

Overall, based on the results obtained, it is possible to state our goals were fully achieved. We were able to build an easy to learn and use system regarding the EO Tool Packaging.

5.3 Workflow Execution

Succeeding the validation of the EO Tool Packaging befalls the Workflow Execution validation. With the EO tool packed we must now provision and configure the cloud instance to deploy and execute the WPS in the workflow chain. On a similar note to the EO Tool Packaging, our system also requires user inputs to fully automates these stages.

5.3.1 Manual Steps

Despite the different approaches presented in the EO Tool Packaging validation, they do not possess a direct impact in the cloud deployment. In other words, the cloud deployment is always composed by the provisioning, configuration and deployment of the WPS.

It is important to bear in mind, the provisioning steps for the validation of the cloud deployment differ from provider to provider. The validation of these process will be done with the AWS and as a cloud provider, since they are the initial cloud set for our system. In both cases, provision is accomplished through the usage of the web browser client. The provision of a cloud instance in AWS can be described through the following steps:

- Select the desired operating system image;
- Select the most optimal hardware configuration;
- Specify the number of instances to only one;
- Use the default VPC, subnet and auto-assign public IP choice;
- Specify the necessary storage;
- Configure the security group to allow SSH and HTTP;
- Click on launch button;
- Create a new key pair;
- Specify the key pair name;
- Download the key pair;
- Click on launch instances button.

Additionally, if the default options are not available for the VPC, extra steps are required besides the creation of these. These defaults already have an internet gateway and routing tables configured. As a result, from the inexistent of the default VPC, it is crucial to create a internet gateway, associating it to the new VPC and configure the rout table. Additionally, if the subnet is missing it is required to create on in the VPC that is going to be used.

While the cloud provisioning is done via web browser, the configuration is done through SSH with the terminal console, only then, users are able to configure the instance. The configuration of the instance is comprised essentially in the installation of Docker, to pull and execute the image created in the EO Tool Packaging, and Nginx. While the users configured the security group in the previous steps to allow HTTP, the WPS is only available locally at the moment. Nginx enables the exposure of this local applications to the world via redirect proxying. Upon installing the Nginx, users need to include in the Nginx configuration file the lines shown in the Listing 5.1, the ip_addr represents the IP where the Docker container is running. Subsequently, restart the Nginx.

```
server{
    listen 80;
    location /{
        proxy_pass "http://{{ ip_addr }}}:5000";
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Fowarded-For $proxy_add_x_forwarded_for;
    }
}
```

Listing 5.1: Nginx configuration file

Therefore, to access and configure the cloud instance the users must:

- Go to the folder containing the downloaded private key;
- Execute the command chmod 400 private-key-name;
- Connect to the cloud instance via SSH
- Update the system;
- Install the Docker;
- Pull the WPS Docker image from the repository;
- Create and run a Docker container using the Docker image of the WPS;
- Install the Nginx;
- Modify the Nginx configuration file;
- Restart the Nginx.

5.3.2 System Steps

As seen in the previous chapter, our system integrated WEFEOS to provide management and validation of workflows. Therefore, the users will utilize WEFEOS to create their own workflow. Every WPS is specified here along with the execution order, the input data and which output is going to be used as input for the WPS. It is important to mention by using the WEFEOS we are restricted to sequential workflows. In other words, since WEFEOS do not possess any gateway, our system was not design with this approach in mind. As a result, our system is less expressive in this step when compared to the manual, since the users are able to execute in parallel if the WPS allows them to. At the end of specifying every information during the workflow creation, the users are now able to verify if everything is correct by pressing the validate button and see if the console log has any error. Henceforth, with the creation and validation of the workflow, the users must provide information regarding not only the cloud provider and region for each WPS present in the workflow but also the workflow engine, the execution mode, size mode and size of the cloud instance they want to utilize. As mention on the previous chapter, we developed our system with the goal of allowing the expansion of our set of workflow engines and cloud providers with more choices in the future. Therefore, our system does not lose expressivity in this regard. It is important to bear in mind, it is only possible to trigger the Workflow Execution after every information is present and the workflow is validated. To summarize, the users require to follow the subsequent steps:

- Specify the WPSs in the dashboard;
- Link all the WPS's in the dashboard;
- Specify the inputs for each WPS;
- Validate the workflow;
- Specify the workflow engine and execution mode;
- Specify the cloud provider and region for each WPS;
- Specify the hardware storage for each WPS;
- Select the submit button.

5.3.3 Errors

In the execution of each step in the cloud deployment procedure the users are susceptible to mistakes. Contrary to the EO Tool Packaging process, when the execution of the procedure is done manually, the errors obtained might not correspond to the execution step in which the users are. In the event of specifying incorrectly the hardware space needed or misconfiguring the security group, the users may only notice this mistake during the configuration of the instance or later when trying to execute the WPS. This mistake would require deleting the instance created and start anew. On the other hand, wrongly configuring the Nginx configuration file or missing inputs for the WPS execution only require redoing the corresponding step. As a result, we present a list of possible errors for the manual execution:

- Select an underpowered hardware configuration;
- Selecting more instances than needed;
- Miss click default options;
- Missing the configuration of the internet gateway or route table if necessary;
- Missing the HTTP configuration on the security group;
- Not selecting enough disc space;
- Forget to update the operating system;
- Misconfigure the Nginx configuration file;
- Pull the wrong Docker image;
- Missing or type incorrectly the inputs for the execution.

With respect to our system, the definition of the workflow is done previously to the execution. Using WEFEOS syntactic validation, we are able to prevent some of the manual errors such mismatching data types between inputs and outputs. However, it is still possible for the WPSs to produce an incorrect output despite its specification. Regarding this occurrence, we are able to detect with the dynamic validation of WEFEOS, stop the whole workflow execution and notify the users. In addition, our system is not able to validate all user inputs. In other words, information regarding hardware configuration specified during the EO Tool Packaging. The correction of these types of errors require to repackage the EO tool and start the process from scratch. Nevertheless, through the automation we can also avoid malformed Nginx configuration file or missing a configuration in the operating system.

All things considered, our system is only susceptible to misinformation regarding hardware configuration given by the users or the unexpected incorrect output obtained.

5.3.4 Execution

In the same vein as the EO Tool Packaging, the validation of the Workflow Execution process is performed by the same users. Additionally, we will also have a learning phase prior to the validation process. The learning phase will occur in the same manner as in the learning phase of the EO Tool Packaging. In this matter the learning is applied to our system only, the workflow manager section. For the same reasons above, we only measure the times regarding our system. Even though the documentation available to deploy a cloud instance and run a Docker container is greater than the PyWPS documentation, the learning and executing procedure would still take days. As a result, it is meaningful to measure it. However, we still describe some comparisons between the manual and the automated approach.

In the Workflow Execution validation, we are going to provide the WPS ourselves in order to chain them, releasing some extra burden on the users from creating a new EO tool and encapsulating them. The workflow for this validation is composed by two of the three WPS's utilized in the previous chapter to calculate the rainfall anomaly in a given time range. The first WPS is responsible for the aggregation, sum, of the total rain in a given time range. The second WPS is responsible for calculating the long term averages of the aggregations, the arithmetic average. With usability as our main focus, it is important to bear in mind we are not validating the execution of the WPS's in the workflow. What we are validating is if the users can build the workflow using our dashboard while expressing every single piece of data relevant for its execution, validation and deployment with ease. The comparison of the workflow execution in both cases is a problem more related with performance. In this field, our system is compelled have better results in the event of the WPS's not being executed in parallel. The time to detect when the execution of a WPS ended is much higher when done manually, especially if the execution takes hours, since the users are not notified and commonly leave this WPS running in background to perform other tasks in the meantime. With the process being automatized, our system gets notified as soon as the execution finished starting right away the deployment of the next WPS. Another major key point that directly has an impact in the performance is the fact that automation by itself is always faster than a person typing and doing task one by one manually. With the increase in WPS's, tasks and complexity the time spent in each step is significantly increase with the manual approach. As a result, we are not going to focus on the execution of the whole workflow. Instead, we are only concerned with the users constructing the workflow in the dashboard, specify the inputs, validate them and state the workflow engine, execution mode, cloud provider, cloud region, size mode and the size required for the instance. Thus, we are going to measure the time spent in this process.

In the same vein as the EO Tool Packaging validation, we will also track the errors performed by the users in the cloud deployment using the list of possible errors described above.

5.3.5 Validation Script

Considering both usability tests are performed to the same users, in this phase we do not use the introductory speech to our system. Instead we proceed to explain what we hope to achieve with our system regarding the execution of the workflow in the cloud. As a result, continuing the validation script above, we augment the list with the following tasks:

- With the EO Tool Packaging task being completed, explain to the user we are going to start validating the Workflow Execution. Present all the documentation required for the execution of the workflow in our system and solicit the user to read it;
- Proceed to explain which WPSs the user is going to utilize, what they accomplish, their inputs and outputs;
- Proceed to request the user to use the Workflow Execution in our system. Compose a workflow with the WPSs provided, validate them and provide the necessary cloud information;

- State to the user we concluded our session and thank him for is participation in the validation of our system.

5.3.6 Results

The validation of the execution of workflows in the cloud environment was accomplished in the same day as the validation of the EO Tool Packaging. As a result, usability tests were done for the same users, in the same week, utilizing the same environment and conditions.

Once the usability test for the Workflow Execution were finished, we grouped the results and displayed them into one table. In the same vein as the EO Tool Packaging, the Table 5.3 contains all the times measured in the validation test, both learning and execution phase. During the execution phase of the Workflow Execution there was no occurrence of any mistakes using our system. It is important to mention both tasks were accomplished with success on the first try.

Learning Phase	Execution Phase
145	652
220	904
205	763

Table 5.3: Workflow Execution times recorded from the system validation

5.3.7 Analysis

In the same manner as the EO Tool Packaging documentation, the Workflow Execution guide is also fairly simple to read. By using real images of the system with described details on what to do and what each field requires, it was possible for the users to read it with ease and fully understand how the system works. As a result, the learning phase per person average 190 seconds. Contrary to the manual approach, which would require a day to discover and fully understand how to perform each step described above.

In addition to the meticulous and simple documentation, the system provides the best tools to enable the users to provide every crucial information in a swift and secure manner. The use of a dashboard, to construct the workflow, supported by a validator along with dropdown menus, to specify the additional information concerning the cloud deployment, assists the users by minimizing possible mistakes. It also important to emphasize that no errors were seen during the usability tests. Moreover, by utilizing a few clicks the users are capable of providing every information without mistakes. Not only does it reduce the complexity of executing a workflow through a cloud environment but also enables the users to spend less time using the system. As result, based on the values obtained in the Table 5.3, the average time to complete this procedure is 773 seconds. As already explained, executing a workflow in the cloud manually would possible takes days. Since the users do not have any mechanism to verify if every value and configuration is correct, they are always prone to errors, incrementing the time required. Additionally, it is required more steps from the users, as stated above and it is required from the users to spend more time verifying when the execution of a WPS finishes to deploy the next cloud instance for the following WPS in the workflow chain. It is important to mention, the longer the workflow, the more time is required. However, the increase in time by doing the procedure manually is significantly more than using our system. As a result, the results obtained by utilizing our system are far greater than doing the process manually.

In essence, we are able to observe every goal was achieved in our system. An accessible system, easy to learn and use regarding the workflow execution in the cloud. While at the same time being able to minimize the possible mistakes that users are susceptible when doing the procedure manually.

Chapter 6

Conclusions

To summarize, the procedure of wrapping EO tools into WPSs and deploying them to the cloud platform is time consuming, repetitive and requires a certain amount of knowledge when done manually. In the end, it places unnecessary burden in the user that can translate into avoidable mistakes and can quickly become cost heavy. Therefore, we aimed to develop continuous delivery pipelines that ease these burdens from the user.

The first pipeline consists in the EO Tool Packaging. We transformed the EO tools on a widely used standard, WPS, through the use of templates. Subsequently, we used Docker to provide an isolated environment with every dependency required for the WPS. With Docker as the encapsulation technology, we opted to use Dockerhub as the main repository to store our images.

The second pipeline supports the Execution of Workflows in the cloud. We utilized WEFEOS for the data validation, preventing resource waste from mismatching, missing or unavailability errors. Through the use of Ansible, we were able to create playbook to automate the provision and configuration of cloud instances to deploy the WPS docker containers. Moreover, Ansible enable our system to have support with multiple cloud providers.

Additionally, we implemented a WPS management system, from several catalogs provided by different Geospatial organizations, to enable the future use of WPS already running in servers.

With the solution described and supported by the validation results, we can state that all proposed objectives of providing a high degree of automation were accomplished with success. Moreover, with the contribution to the OG engineering report it is possible to observe that indeed our system has a positive impact in the EO community.

6.1 Achievements

In the light of this thesis, the major achievement, regarding the impact in the EO community, is the automation and safe execution of workflows using cloud environments and the transformation of EO tools into WPS. Additionally, it is also important to mention the following achievements:

- A platform flexible enough to enable future addition of catalogs for WPSs discoveries as well as workflow engines and cloud providers for the execution of workflows;

- A system that enables the users to manage their WPSs in the different catalogs;
- An environment where users are able to share their EO tools among each other's.

6.2 Limitations

As it is observe along the chapter 4 and the chapter 5, the automation of the EO tool Packaging comes with some minor limitations. Even though these limitations do not present a real impact, they are still worth of mention. Additionally, the Workflow Execution also presents a limitation worth of being solved in the future.

With the automation of the EO Tool Packaging, it is lost some expressivity regarding the errors occurred during the WPS execution. To solve this problem, we log in a text file all the details WPS catch during its execution. This text file is available through a REST Endpoint path named logs. However, due to automatization process, it is not possible to obtain information regarding the malfunction of the EO tool. If the EO tool has a bug, we are not able to detect the line of code where it occurred and report back to the user. Contrary to the manual packaging, it is possible to report these errors back, since the code is tailored specifically for the WPS. Every other type of error does not pose a limitation since it is possible to detect and record.

As already explained, the other possible limitation is the ComplexOutputData. In other words, our system is only able to detect outputs in the form of files. LiteralData or BoundingBoxData are provided by the console. It is not feasible to parse the information to obtain them, since there might be several of these outputs in a single EO Tool. Contrary to the automation, through manual approach it is possible to specify them a priory. However, in the EO field the EO tools only produces ComplexData types as a norm.

Regarding the Workflow Execution, the crucial limitation comes from the lack of parallelism in the system. WEFEOS was not design with parallelism in mind. As a result, its JSON specification and schema are not able to validate it. In addition, the dashboard is lacking the specification to enable the construction of the workflow.

6.3 Future Work

As referenced above, our system is bound to the dashboard provided by the WPS-V-WEB. As a result, our system only supports sequential workflow executions. Further developing the dashboard and the system to provide parallel execution brings numerous advantages. The execution of workflows where the execution time is a constrain greatly benefit from using parallelism. An example of where time is crucial when using these workflows to obtain information is in case of natural disasters. A more concrete example is the execution of a workflow regarding a volcanic eruption. The scientists and the national disaster response team will be interested to measure ground deformation due to seismic movements. A processing chain will be triggered to detect changes in the critical structures, in particular buildings, for risk assessment. By executing several WPS's simultaneously, it decreases the time spent executing these workflows. Moreover, by introducing parallelism to the system, it enables the possibility to divide the execution of a WPS processing a dataset between several cloud instances while executing them at the same time. Thus, achieving faster workflow execution times.

Throughout the development of our system, we found the monetary costs are affected by numerous variables when deploying in the cloud. One of these variables is the hardware configuration. In other words, how much RAM and CPU's is required to execute the WPS. Choosing the right hardware configuration might reveal to be a slightly tricky. Either the resources are not enough and the process end up failing, or too many resources were selected and not fully utilized. Additionally, the monetary cost for the hardware configuration not only varies from cloud provider to cloud provider, but also from region to region. Another key factor that has an impact in the monetary cost is the outbound and inbound data transfer. Transferring data from a server to a cloud instance or from cloud provider to cloud provider comes with additional charges depending on the data size and transmission time. With all these variables available, users might feel overwhelmed, resulting in not taking the best approaches. Therefore, it is important to enhance the system with a machine learning module. By taking into consideration these variables, it is possible to create a module to use machine learning to calculate and recommend the best decision to make.

Overall, the introduction of both parallelism and machine learning module into the system, would greatly benefit the users. Not only it would improve and expand the workflow execution but also reduce the monetary cost. Additionally, both implementations would increase the users satisfaction.

Bibliography

[1] M. Mihailescu and Y. M. Teo. Dynamic resource pricing on federated clouds. *CCGrid 2010 - 10th IEEE/ACM International Conference on Cluster, Cloud, and Grid Computing*, pages 513–517, 2010. doi: 10.1109/CCGRID.2010.123. URL `https://ieeexplore.ieee.org/document/5493446`.

[2] R. Buyya, C. Shin Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25:599–616, 2009. doi: 10.1016/j.future.2008.12.001. URL `www.elsevier.com/locate/fgcs`.

[3] L. Zhu, L. Bass, and G. Champlin-Scharff. DevOps and Its Practices. *IEEE Software*, 33(3): 32–34, 2016. ISSN 07407459. doi: 10.1109/MS.2016.81. URL `https://ieeexplore.ieee.org/document/7458765`.

[4] J. Wettinger. Gathering Solutions and Providing APIs for their Orchestration to Implement Continuous Software Delivery. Master's thesis, Universität Stuttgart, 2017. URL `https://elib.uni-stuttgart.de/handle/11682/9110`.

[5] P. C. Manuel J. Fonseca and D. Gonçalves. *Introdução ao Design de Interfaces*. FCA, 2012. ISBN 9789727227389.

[6] D. Norman Lee Faus. Packaging an Application. Technical Report US 9323519B2, Red Hat Inc, 2016. URL `https://patents.google.com/patent/US9323519B2/en`.

[7] E. C. Bailey. *Maximum RPM Taking the RPM Package Manager to the Lim-it*. Red Hat, Inc., 2000. ISBN 1888172789.

[8] M. J. Scheepers. Virtualization and Containerization of Application Infrastructure: A Comparison. Technical report, 2014. URL `http://mmc.geofisica.unam.mx/acl/Herramientas/MaquinasVirtuales/VirtualizacionEnLinuxCon-Containers/539ae779eb69a.pdf`.

[9] J. Turnbull and C. Pahl. Containerization and the PaaS Cloud. *Published by the IEEE Computer Society*, 7(11):24–31, 2015. ISSN 23256095. doi: 10.1109/MCC.2015.51. URL `https://www.computer.org/csdl/magazine/cd/2015/03/mcd2015030024/13rRUyoPSR7`.

[10] J. Turnbull. *The Docker Book*. Turnbull Press, 2014. ISBN 9780988820203.

[11] C. Lueninghoener. Getting Started with Configuration Management. page 17, 2011. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.193.9608&rep=rep1&type=pdf.

[12] M. Mohaan and R. Raithatha. *Learning Ansible*. Packt Publishing Limited, 2014. ISBN 9781783550630.

[13] Y. Brikman. *Terraform Up and Running Writing Infrastructure as Code*. O'Reilly Media, 2017. ISBN 9781491977088.

[14] Matthias Mueller and Benjamin Pross. OGC WPS 2.0.2 Interface Standard. Technical Report 14-065r2, Open Geospatial Consortium, 2015. URL http://docs.opengeospatial.org/is/14-065/14-065.html.

[15] B. Schaeffer. Towards a Transactional Web Processing Service (WPS-T), 2013. URL https://wiki.52north.org/pub/Geoprocessing/TransactionalWPS/Schaeffer_-_Towards_a_Transactional_Web_Processing_Service.pdf.

[16] C. Z. Charles Chen, Tom Landry, Ziheng Sun. OGC Testbed-13: Cloud ER. Technical Report OGC 17-035, Open Geospatial Consortium, 2018. URL http://docs.opengeospatial.org/per/17-035.html.

[17] P. S. Pedro Gonçalves, Panagiotis A. Vretanos, Patrick Jacques, Christophe Noël. OGC Testbed-13: EP Application Package ER. Technical Report OGC 17-023, Open Geospatial Consortium, 2018. URL http://docs.opengeospatial.org/per/17-023.html.

[18] P. S. Pedro Gonçalves, Peter Vretanos, Patrick Jacques, Christophe Noël. OGC Testbed-13: Application Deployment and Execution Service ER. Technical Report OGC 17-024, Open Geospatial Consortium, 2018. URL http://docs.opengeospatial.org/per/17-024.html.

[19] D. Rafael Ferreira Lopes. Workflow Engine for Earth Observation Services. Master's thesis, Instituto Superior Técnico, 2018. URL https://fenix.tecnico.ulisboa.pt/downloadFile/563345090416386/79018-Diogo-Ferreira-Thesis.pdf.

Appendix A

XML Describe Process

We will present an example of a XML resulting from a Descibe Process request of the SayHello process

```
<wps:ProcessDescriptions xml:lang="en-US" version="1.0.0" service="WPS"
xsi:schemaLocation="http://www.opengis.net/wps/1.0.0../wpsDescribeProcess_response.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xlink="http://www.w3.org/1999/xlink" xmlns:ows="http://www.opengis.net/ows/1.1"
xmlns:wps="http://www.opengis.net/wps/1.0.0">
    <ProcessDescription statusSupported="true"
    storeSupported="true"
    wps:processVersion="1.3.3.7">
        <ows:Identifier>say_hello</ows:Identifier>
        <ows:Title>Process Say Hello</ows:Title>
        <ows:Abstract>Returns a literal string output with
        Hello plus the inputed name</ows:Abstract>
        <DataInputs>
            <Input maxOccurs="1" minOccurs="1">
                <ows:Identifier>name</ows:Identifier>
                <ows:Title>Input name</ows:Title>
                <ows:Abstract/>
                <LiteralData>
                    <ows:DataType
                    ows:reference="http://www.w3.org/TR/xmlschema-2/#string">
                        string
                    </ows:DataType>
                    <ows:AnyValue/>
                </LiteralData>
            </Input>
        </DataInputs>
```

```xml
        <ProcessOutputs>
            <Output>
                <ows:Identifier>response</ows:Identifier>
                <ows:Title>Output response</ows:Title>
                <ows:Abstract/>
                <LiteralOutput>
                    <ows:DataType
                    ows:reference="http://www.w3.org/TR/xmlschema-2/#string">
                        string
                    </ows:DataType>
                </LiteralOutput>
            </Output>
        </ProcessOutputs>
    </ProcessDescription>
</wps:ProcessDescriptions>
```

Listing A.1: SayHello Describe Process XML

Appendix B

Documentation

We present here every piece of documentation used by the users to learn how to use our system.

B.1 MCDEEOS Guide

Welcome! We are here to present you MCDEEOS. A system that enables you to use cloud resources to execute Earth Observation (EO) Tool. Through this guide we will teach you how to:

1. Package an EO Tool;

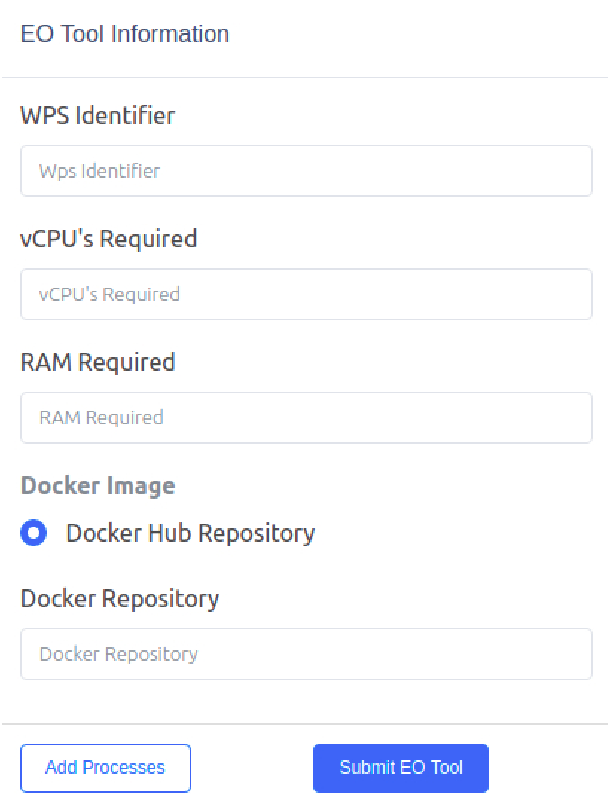
2. Execute a Workflow;

B.1.1 EO Tool Packaging

EO Tool Packaging is a feature that encapsulates any given EO Tool into a Docker Image. In order to expose these EO Tools to the web context we use the OGC interface standard Web Processing Service (WPS). In other words, we wrap the EO Tool into a WPS and then encapsulate it. In the end we make available the resulting image in the Dockerhub. Pré-requisites:

1. EO Tool already encapsulated with all the dependencies in a Docker image and available at Dockerhub.

We will start by describing every information required for each field of the EO Tool Information Section, figure B.1.


EO Tool Information

WPS Identifier

Wps Identifier

vCPU's Required

vCPU's Required

RAM Required

RAM Required

Docker Image

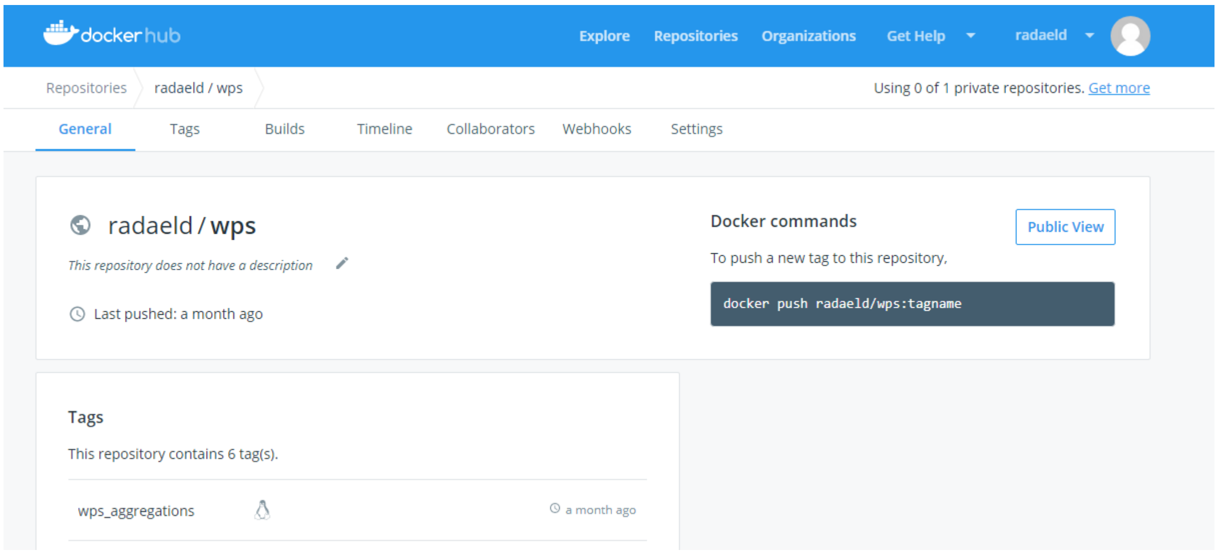
● Docker Hub Repository

Docker Repository

Docker Repository

Figure B.1: EO Tool Information

- WPS Identifier – corresponds to the name of the WPS resulting from the encapsulation process;
- vCPU – number of CPU's necessary for the EO tool to work perfectly;
- RAM – number of RAM necessary for the EO tool to work perfectly;
- Docker Repository – the name of the Dockerhub repository. The format of the name is dockerhub_username/repository_name:tagname. Ex: radaeld/wps:wps_aggregations. This information can be seen in the Dockerhub website as shown in the figure B.2.
- Add Processes Button – adds an extra Process information box;
- Submit EO Tool Button – submit the EO Tool for the packaging process.



Using 0 of 1 private repositories. Get more

radaeld / wps

This repository does not have a description

Last pushed: a month ago

Docker commands

To push a new tag to this repository,

```
docker push radaeld/wps:tagname
```

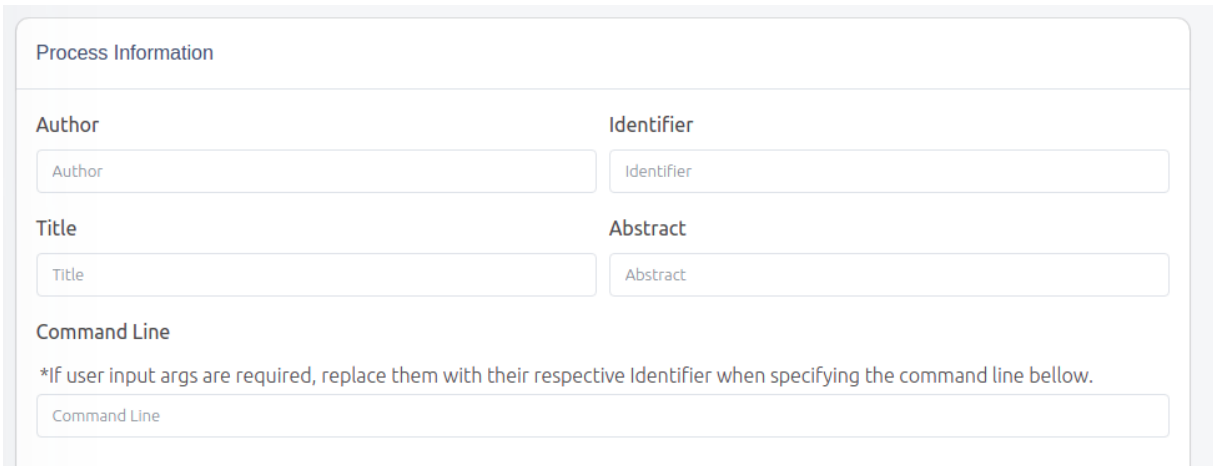
Tags

This repository contains 6 tag(s).

wps_aggregations — a month ago

Figure B.2: Dockerhub Repository

With the EO Tool Information described, we will now start with the Process information Section, figure B.3.



*If user input args are required, replace them with their respective Identifier when specifying the command line bellow.

Figure B.3: Process Information

- Author – name of the user that created the EO Tool;
- Identifier – identifier for the EO Tool process;
- Title – title of the EO Tool process;
- Abstract – description of the EO tool process, what it accomplishes;
- Command Line – the normal command line for the execution of the program. However, if it requires an input as an argument, the user must use its identifier, that will be defined bellow, in here to specify. Ex: we have a python program that gives as an output a Hello World! and the name of this file is helloworld.py. In this field we would type python helloworld.py. Now lets suppose instead of Hello World it would take as an input our name and give it as an output. Imagine my name is Carlos, it would respond Hello Carlos! In this case we type python hello.py UserName. Where we specified bellow using the LiteralInputData button to add a literal data input box and in the Identifier field we would have filled with UserName in it. Moreover, the user can specify flags used by the program in here.

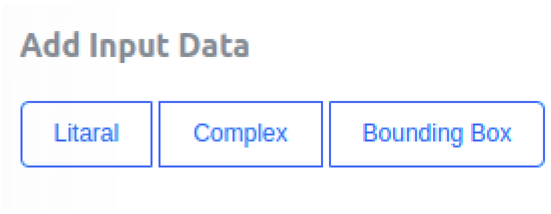
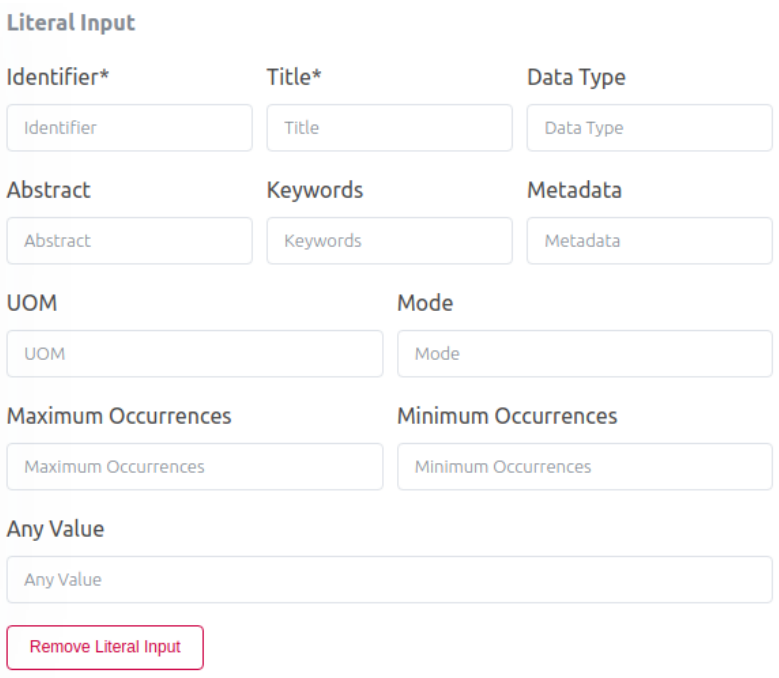


Figure B.4: Input Data Buttons

The figure B.4 shows the buttons to add a box with the corresponding data types of the inputs required for the EO tool process. The mandatory fields for each data type are marked with a * in their name. They can be summarized in the following:

- LiteralInput – identifier, title;
- ComplexInput – identifier, title and default format. Inside the default format the mime type must be specified;
- BoundingBoxInput – identifier, title and crss.

The figure B.5 above represents the Literal Input Data standard for the EO tool. The Identifier is name of the input data field. In the example of the Command Line, we would type in Username in this identifier textbox. Literal Input Fields:


Literal Input

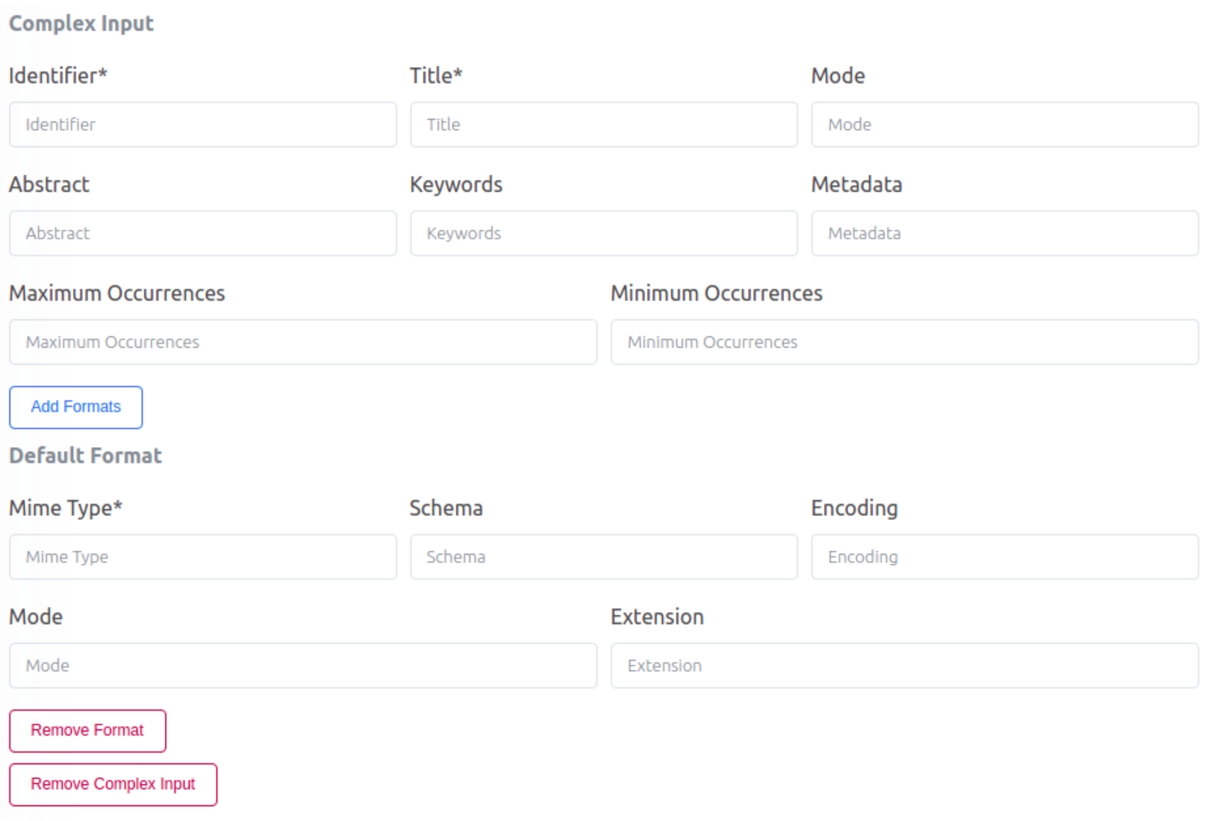
Identifier*	Title*	Data Type
Identifier	Title	Data Type
Abstract	Keywords	Metadata
Abstract	Keywords	Metadata
UOM	Mode	
UOM	Mode	
Maximum Occurrences	Minimum Occurrences	
Maximum Occurrences	Minimum Occurrences	
Any Value		
Any Value		

Figure B.5: Literal Input Data

- Identifier – identifier of the literal input;
- Title – the title of the input;
- Data Type – the type of the data. Default value is integer;
- Abstract – input abstract. Default value is ‘’;
- Keywords – keywords that characterize the input. Default value is [];
- Metadata – metadata about the input. Default value is [];
- UOM – units of the input. Default value is None;
- Mode – validation mode. None = 0, Simple = 1, Strict = 2, Very Strict = 3. Default value is 0;
- Maximum Occurrences – maximum occurrences of this input. Default value is 1;
- Minimum Occurrences – minimum occurrences of this input. Default value is 1;

- Allowed Value – only values allow for this input. Default value is None;

The Remove Literal Input button removes the literal input box from the process information.
The figure B.6 above represents the Complex Input Data standard for the EO tool.


Complex Input

Identifier*	Title*	Mode
Identifier	Title	Mode
Abstract	Keywords	Metadata
Abstract	Keywords	Metadata
Maximum Occurrences	Minimum Occurrences	
Maximum Occurrences	Minimum Occurrences	

Default Format

Mime Type*	Schema	Encoding
Mime Type	Schema	Encoding
Mode	Extension	
Mode	Extension	

Figure B.6: Complex Input Data

Complex Input Fields:

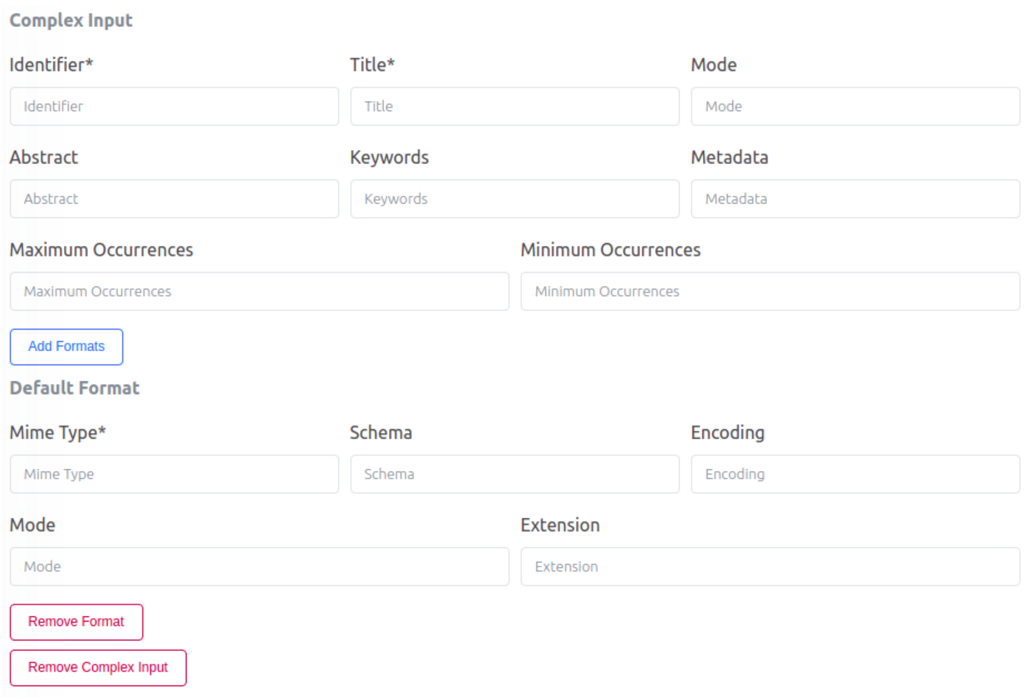
- Identifier – identifier of the literal input;
- Title – the title of the input;
- Abstract – input abstract. Default value is '';
- Keywords – keywords that characterize the input. Default value is [];
- Metadata – metadata about the input. Default value is [];
- Mode – validation mode. None = 0, Simple = 1, Strict = 2, Very Strict = 3. Default value is 0;
- Maximum Occurrences – maximum occurrences of this input. Default value is 1;
- Minimum Occurrences – minimum occurrences of this input. Default value is 1;

Add format button enables the addition of several formats for the same Complex Data Input.
Default Format and Supported Format Fields:

- Mime Type – mime type definition;
- Schema – XML schema definition. Default value is None;
- Encoding – base64 or not. Default value is None;
- Mode – validation mode. None = 0, Simple = 1, Strict = 2, Very Strict = 3. Default value is 0;
- Extension – file extension. Default value is None.

The Remove Format button removes the format box out of the Complex Input. However, it is necessary at least one format, the default. The Remove Complex Input button removes the literal input box from the process information.

The figure B.7 above represents the Bounding Box Input Data standard for the EO tool.


Complex Input

Identifier*	Title*	Mode
Identifier	Title	Mode
Abstract	Keywords	Metadata
Abstract	Keywords	Metadata

Maximum Occurrences	Minimum Occurrences
Maximum Occurrences	Minimum Occurrences

Default Format

Mime Type*	Schema	Encoding
Mime Type	Schema	Encoding

Mode	Extension
Mode	Extension

Figure B.7: BoundingBox Input Data

BoundingBox Input Fields:

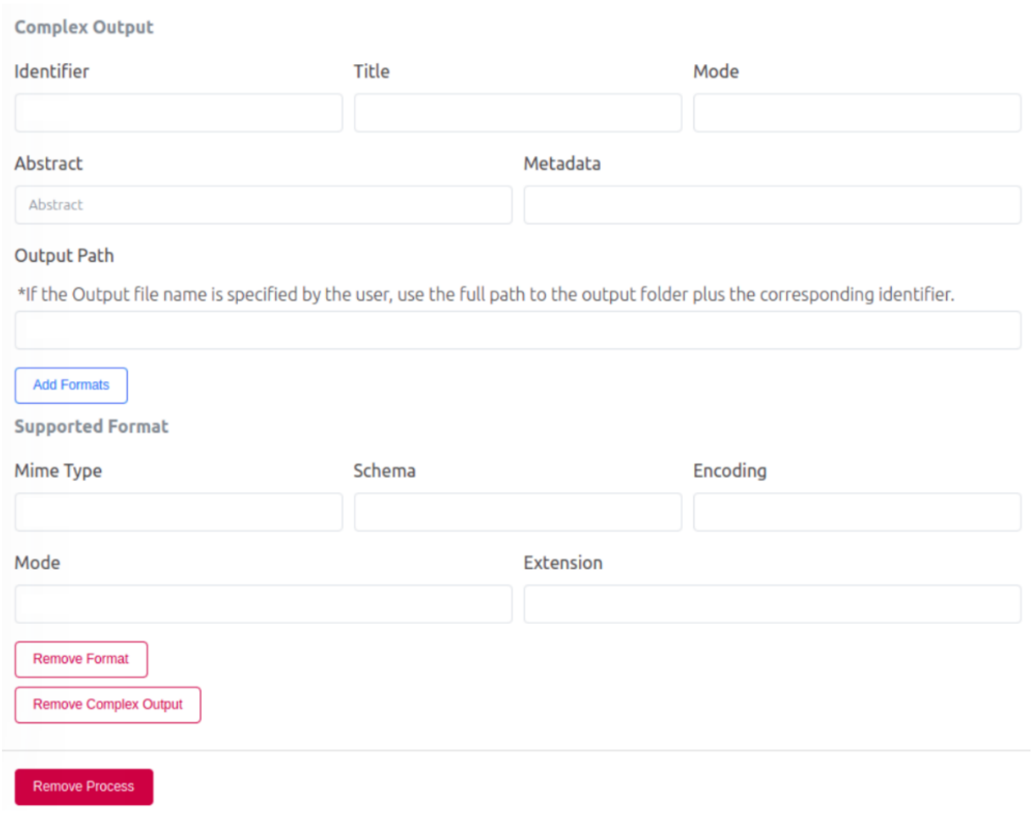
- Identifier – identifier of the literal input;
- Title – the title of the input;
- Abstract – input abstract. Default value is ‘’;
- Keywords – keywords that characterize the input. Default value is [];
- Metadata – metadata about the input. Default value is [];
- Mode – validation mode. None = 0, Simple = 1, Strict = 2, Very Strict = 3. Default value is 0;
- Maximum Occurrences – maximum occurrences of this input. Default value is 1;

- Minimum Occurrences – minimum occurrences of this input. Default value is 1;
- Dimensions – either 2 or 3;
- CRSS – list of supported coordinate reference system. For example ['EPSG:4326'].

Figure B.8: Output Data Buttons

Figure B.8 represents the Complex Output Data button to adds an extra box for describing the EO tool ComplexOutput.

The required fields in the output are the same as in the input. However, an extra field is required, the Output Path. In this, we must specify where the ComplexOutput file is inside the docker image. Ex: the EO process creates an image and for the ComplexOutput identifier we typed Img. If during the development no specific location was specified for the storing, meaning it generates in the folder where the program is executed we just type ./Img. If the image is generated to a specific folder named outputs we must specify the full path starting from the root folder. The root folder in the docker image is /. As a result, we would type /outputs/Img.


Complex Output

Identifier | Title | Mode

Abstract | Metadata

Output Path

*If the Output file name is specified by the user, use the full path to the output folder plus the corresponding identifier.

Supported Format

Mime Type | Schema | Encoding

Mode | Extension

Figure B.9: Complex Output Data

B.1.2 Workflow Execution

Workflow Execution is a feature that enables the creation of a workflow. Furthermore, it executes each WPS composing the workflow in the different cloud providers.

1. EO Tools previously packed by our system in order to use them.

To construct the workflow, we provide a drag and drop dashboard, figure B.10, where the circle is the starting event, the double circle is the end event, the square is the WPS box, the folder the load workflow and the arrow the download workflow.

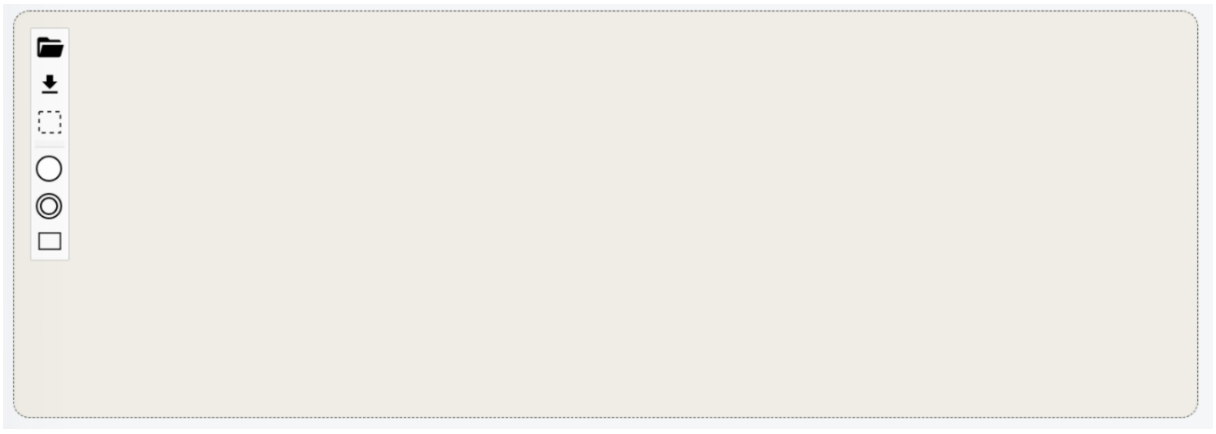
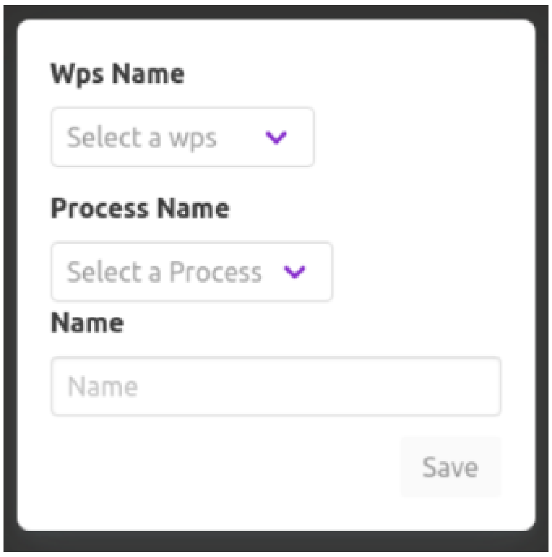


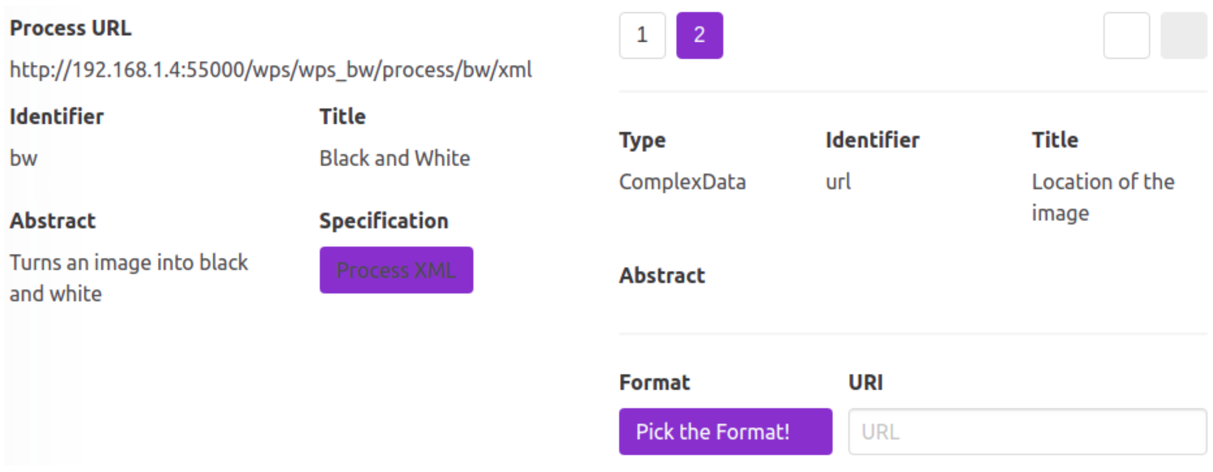
Figure B.10: Workflow Dashboard

After dragging the square box to represent a WPS in the chain a new box will pop up, figure B.11, to fill with a dropdown menu for every detail required of the WPS.



- WPS Name – name of the WPS to use;
- Process Name- name of the processes in the WPS selected to use;
- Name – name for the box representing the WPS in the dashboard.

Figure B.11: Information for creating the WPS in the dashboard


Process URL

http://192.168.1.4:55000/wps/wps_bw/process/bw/xml

Identifier	Title
bw	Black and White

Abstract	Specification
Turns an image into black and white	Process XML

Type	Identifier	Title
ComplexData	url	Location of the image

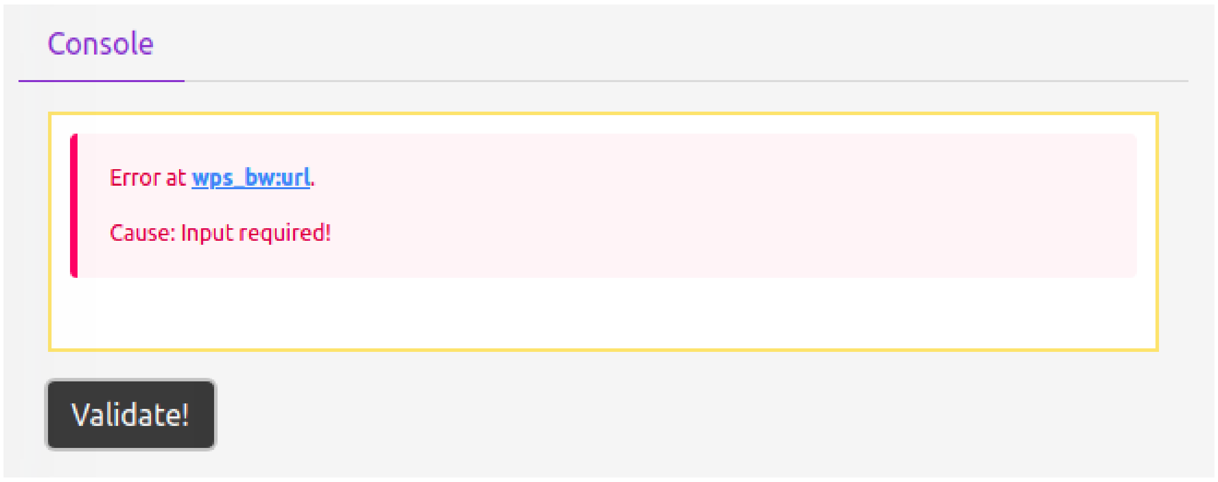
Abstract

Format	URI
Pick the Format!	URL

Figure B.12: Process Information of the WPS

After constructing the workflow, the user must click in every WPS in the dashboard to provide the necessary information requiring the input location and formats, figure B.12.

With everything specified the user is now able to validate the workflow. If an error occurs, the WPS box with an error will be red and the error is displayed in the console log, figure B.13. Otherwise, the workflow in the dashboard will be highlighted in green.

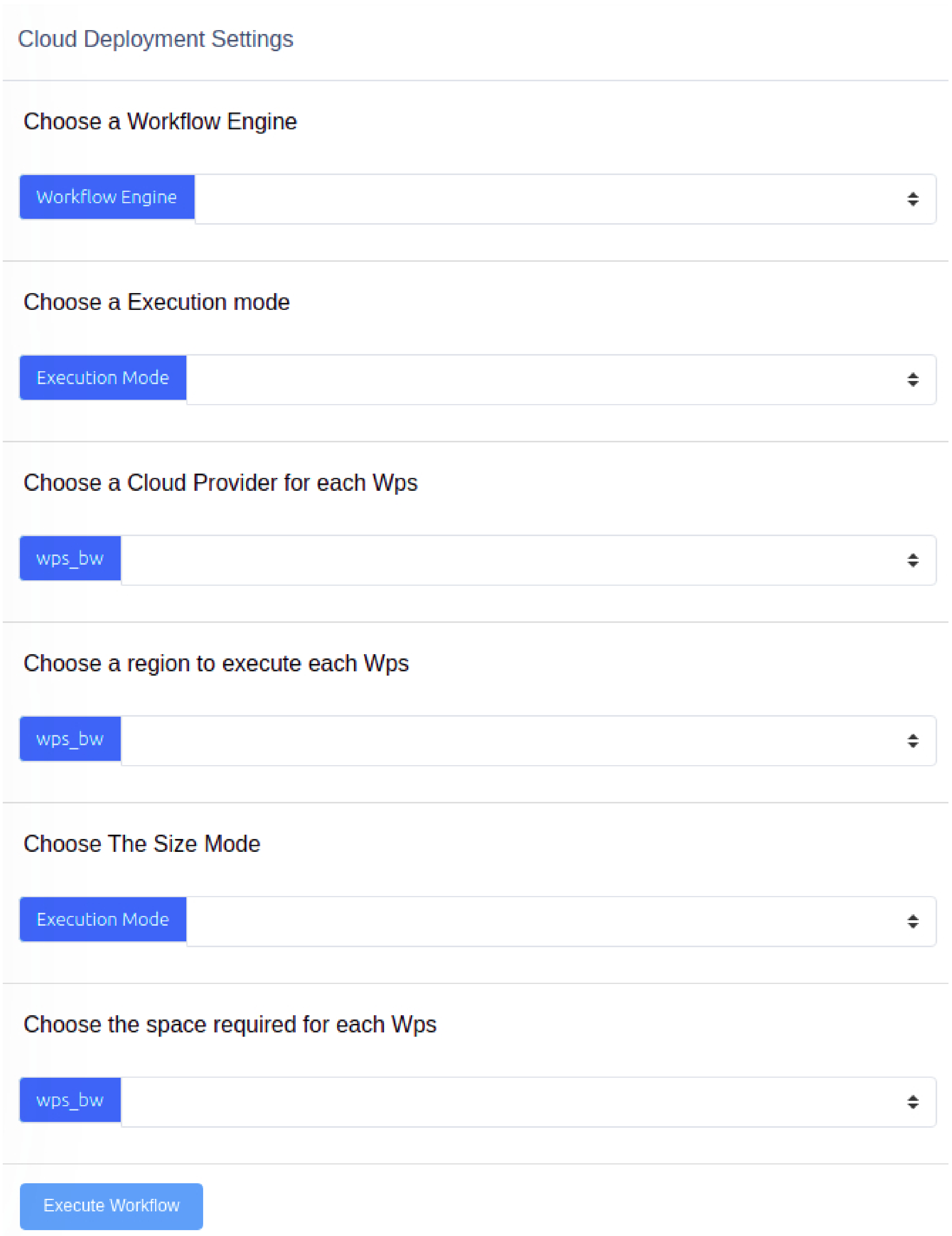

Console

Error at wps_bw:url.

Cause: Input required!

Figure B.13: Console to display validation errors

Once the validation of the workflow is successful, we can select the cloud deployment settings represented in figure B.14. Every information is provided in here through dropdown menus to ease the procedure.



Cloud Deployment Settings

Choose a Workflow Engine

Choose a Execution mode

Choose a Cloud Provider for each Wps

Choose a region to execute each Wps

Choose The Size Mode

Choose the space required for each Wps

Figure B.14: Cloud information required for the workflow execution

Cloud Information details:

- The workflow engine represents the workflow engine that is going to be used;

- The execution mode consists in how the user wants to group the WPSs per cloud instances. For this we present three possible solutions. One WPS per cloud instance. Or we group every WPS if they are in the same cloud and region, if the have the same hardware requirements (TAM and CPU) and deploy them in the same instance. There is a possibility of also grouping every WPS on the same cloud and region and make them available in the same instance even if they have different hardware requirements. However, in this mode, the cloud instance will be created with the higher WPS hardware requirement needed in order for its success;
- The cloud provider is going to dictate the cloud where the WPS will be deployed;
- The cloud region is going to dictate the region available in the cloud where the WPS will be deployed;
- The size mode represents how much space the cloud instance is going to be created with. We present two modes. One to use the specific size the user selects. While the other the user will select the minimum space required and the system will automatically create the instance with the maximum size available by have that value into consideration as the bare minimum.
- The size simple stating the size required per WPS.

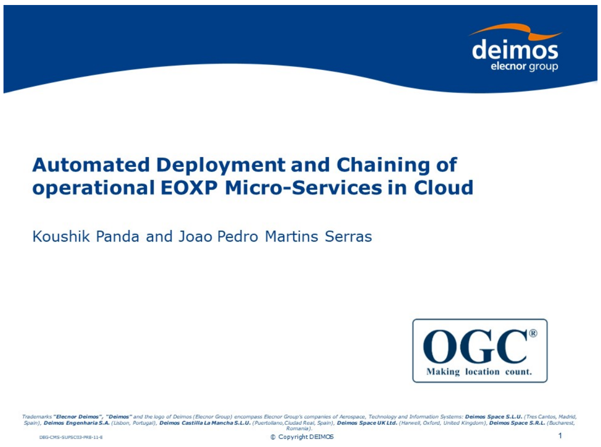
Once every information is specified the user can execute the workflow in the cloud.

Appendix C

OGC Technical Committee Meetings

To ensure our collaboration in the OGC Testbed15 we had to do several meetings throughout the 2019. Therefore, we present the images of the presentations used during these meetings.

C.1 24 January 2019

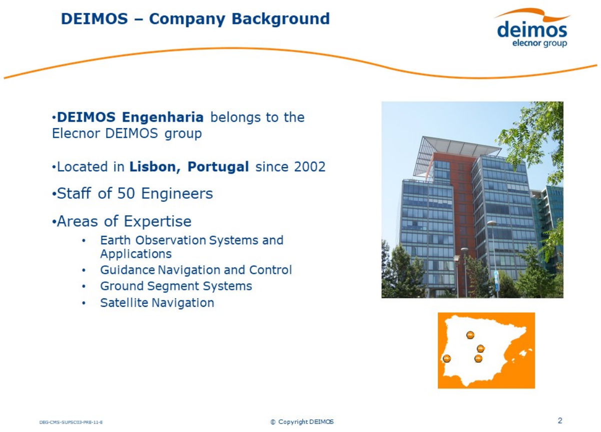


Automated Deployment and Chaining of operational EOXP Micro-Services in Cloud

Koushik Panda and Joao Pedro Martins Serras

*Trademarks **"Elecnor Deimos"**, **"Deimos"** and the logo of Deimos (Elecnor Group) encompass Elecnor Group's companies of Aerospace, Technology and Information Systems: **Deimos Space S.L.U.** (Tres Cantos, Madrid, Spain), **Deimos Engenharia S.A.** (Lisbon, Portugal), **Deimos Castilla La Mancha S.L.U.** (Puertollano,Ciudad Real, Spain), **Deimos Space UK Ltd.** (Harwell, Oxford, United Kingdom), **Deimos Space S.R.L.** (Bucharest, Romania).*

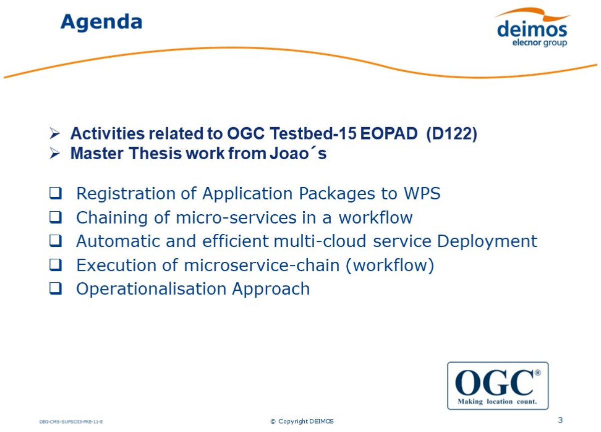
DEG-CMS-SUPSC03-PRE-11-E © Copyright DEIMOS 1



DEIMOS – Company Background

- **DEIMOS Engenharia** belongs to the Elecnor DEIMOS group
- Located in **Lisbon, Portugal** since 2002
- Staff of 50 Engineers
- Areas of Expertise
  - Earth Observation Systems and Applications
  - Guidance Navigation and Control
  - Ground Segment Systems
  - Satellite Navigation

DEG-CMS-SUPSC03-PRE-11-E © Copyright DEIMOS 2

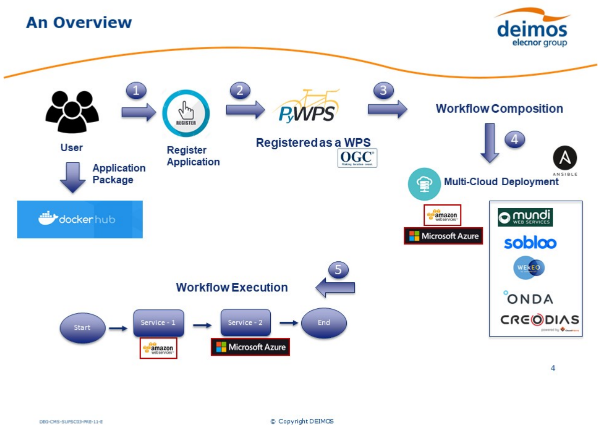


Agenda

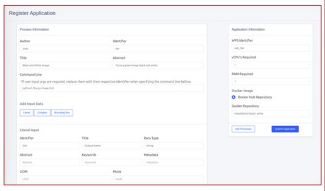
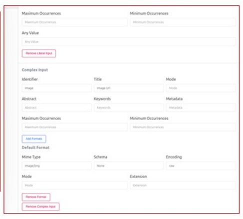
- ➢ **Activities related to OGC Testbed-15 EOPAD (D122)**
- ➢ **Master Thesis work from Joao´s**

- ❑ Registration of Application Packages to WPS
- ❑ Chaining of micro-services in a workflow
- ❑ Automatic and efficient multi-cloud service Deployment
- ❑ Execution of microservice-chain (workflow)
- ❑ Operationalisation Approach

DEG-CMS-SUPSC03-PRE-11-E © Copyright DEIMOS 3

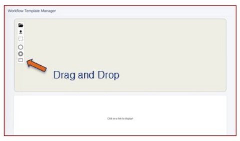


Application Registration

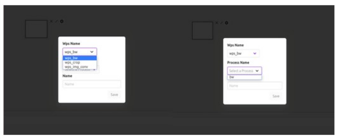



- The Application Developer maintains the application package in the Docker Repository
- Register Application
  - Application owners provided the necessary information
  - Creates the WPS and made available in the Docker Hub
  - The WPS is made available for the Workflow Manager

Workflow Composer



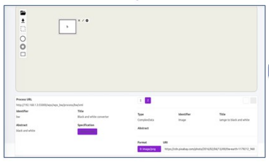
Workflow Composer



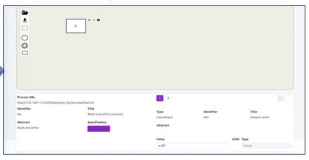
- Configuring the processing modules or Services
  - Identify the Registered WPS
  - User Defined processing module name
  - Validation of the WPS before being included in the workflow – WPS Schema compliant with OGC Standards

Configuring Input and Output of the Service 1 ("b")

Step 1



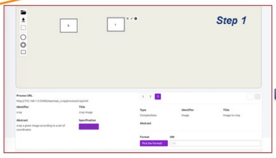
Step 2



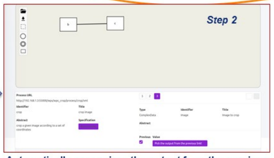
Configuring the Input Data
- Providing Reference to the Input Data
- Defining the type of the Data

Configuring the Output
Creating output Data Identifier
Defining the type of the Data

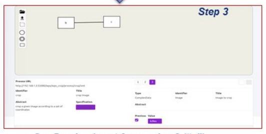
Configuring Input and Output of the Service 2 ("c")



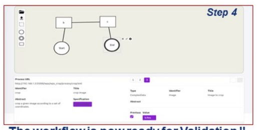
Configuring input for service 2 ("c")



Automatically recognizes the output from the previous service ("b")

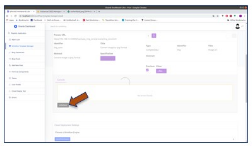


Configuring input for service 2 ("c")



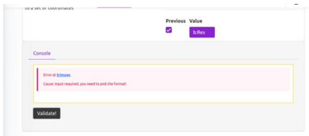
The workflow is now ready for Validation !!

Workflow Validation

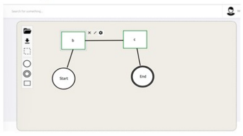


Validate the workflow as a whole before execution

Example of Validation failed

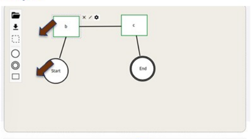


Example of Validation succeeded



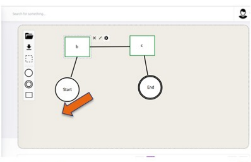
Cloud Deployment and Execution setup

Step 1:



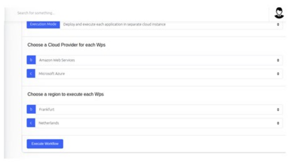
- Flexible options to choose Workflow Engine
  - Currently Internally developed
  - Incorporate Camunda **(In Progress)**
- **Execution mode** (Deploy and Execute)
  - The same Cloud Instance
  - Separate Cloud Instances
  - Cloud Instance with more resources

Step 2:



- **Use Case** : Deploy and execute services in separate Clouds
- Flexible options to choose Cloud Providers for each services

Cloud Deployment and Execution setup



Search for something...

Execution Mode: Deploy and execute each application in separate cloud instance

Choose a Cloud Provider for each Wps

b	Amazon Web Services
c	Microsoft Azure

Choose a region to execute each Wps

b	Frankfurt
c	Netherlands

Execute Workflow

➢ Options to choose the Region for each cloud provider to deploy your application

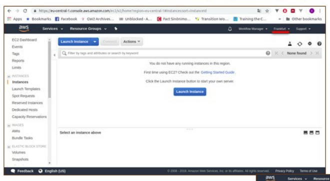
Ready to Deploy and Execute !!

What to expect now :

Create an Instance in AWS and Deploy application "b" → Execute Application "b" and Validate → Create an Instance in Azure and Deploy application "c" → Execute Application "c" and Validate → Terminate the AWS instance

Workflow Execution

Create an Instance in AWS and Deploy application "b" → Execute Application "b" and Validate → Create an Instance in Azure and Deploy application "c" → Execute Application "c" and Validate → Terminate the AWS instance



https://eu-central-1.console.aws.amazon.com/ec2/v2/home?region=eu-central-1#Instances:sort=instanceId

Services · Resource Groups · Workflow Manager · Frankfurt · Support

EC2 Dashboard, Events, Tags, Reports, Limits, INSTANCES, Instances, Launch Templates, Spot Requests, Reserved Instances, Dedicated Hosts, Capacity Reservations, IMAGES, AMIs, Bundle Tasks, ELASTIC BLOCK STORE, Volumes, Snapshots

Launch Instance · Connect · Actions

Filter by tags and attributes or search by keyword — None found

You do not have any running instances in this region.

First time using EC2? Check out the Getting Started Guide.

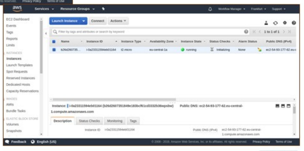
Click the Launch Instance button to start your own server.

Launch Instance

Select an instance above

Feedback · English (US) · © 2008 - 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved. · Privacy Policy · Terms of Use

1 Before workflow Execution there is no AWS Instance !!



Launch Instance · Connect · Actions

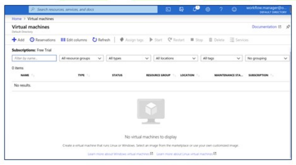
Instance: i-0a23311594eb01164 ... 1.compute.amazonaws.com

Workflow Execution trigger:

2
- **AWS Instance created**
- **Application deployed**
- **Application Execution**

Workflow Execution

Create an Instance in AWS and Deploy application "b" → Execute Application "b" and Validate → Create an Instance in Azure and Deploy application "c" → Execute Application "c" and Validate → Terminate the AWS instance



Search resources, services, and docs · workflow.manager@o... DEFAULT DIRECTORY

Home > Virtual machines

Virtual machines

Default Directory

Add · Reservations · Edit columns · Refresh · Assign tags · Start · Restart · Stop · Delete · Services

Subscriptions: Free Trial

Filter by name... · All resource groups · All types · All locations · All tags · No grouping

0 items

NAME	TYPE	STATUS	RESOURCE GROUP	LOCATION	MAINTENANCE STA...	SUBSCRIPTION
No results.						

No virtual machines to display

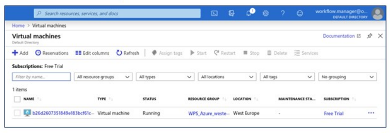
Create a virtual machine that runs Linux or Windows. Select an image from the marketplace or use your own customized image.

Learn more about Windows virtual machines · Learn more about Linux virtual machines

1 Before workflow Execution there is no Azure Instance !!

Workflow Execution trigger

2
- **Azure Instance created**
- **application deployed**
- **Application Execution**



Search resources, services, and docs · workflow.manager@o... DEFAULT DIRECTORY

Home > Virtual machines

Virtual machines

Default Directory

Add · Reservations · Edit columns · Refresh · Assign tags · Start · Restart · Stop · Delete · Services

Subscriptions: Free Trial

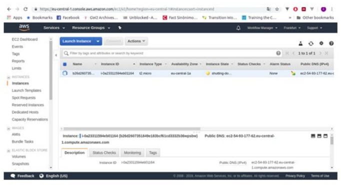
Filter by name... · All resource groups · All types · All locations · All tags · No grouping

1 items

NAME	TYPE	STATUS	RESOURCE GROUP	LOCATION	MAINTENANCE STA...	SUBSCRIPTION
b26d2607351849e183bcf61c...	Virtual machine	Running	WPS_Azure_weste...	West Europe	-	Free Trial

Workflow Execution

Create an Instance in AWS and Deploy application "b" → Execute Application "b" and Validate → Create an Instance in Azure and Deploy application "c" → Execute Application "c" and Validate → Terminate the AWS instance



Launch Instance · Connect · Actions

Instance: i-0a23311594eb01164 ... 1.compute.amazonaws.com

After the successful execution of the workflow, the AWS Instance is terminated

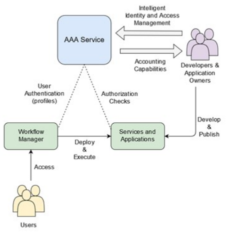
In a nutshell…

Instantiate a new VM in AWS → Deploy and execute Application "b" → Validate the Result → Instantiate a new VM in Azure → Deploy and Execute Application "c" → Terminate the Instance in AWS and Azure

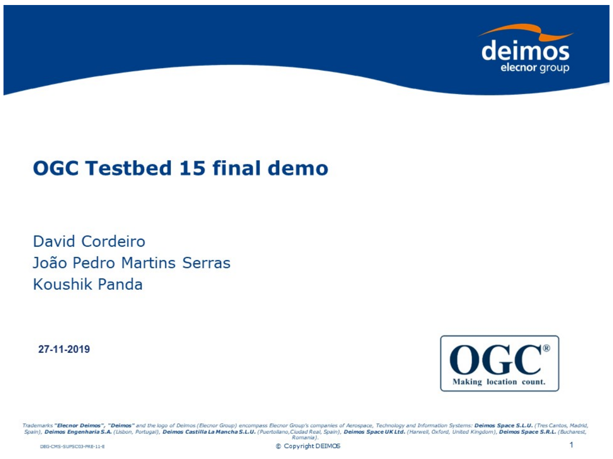
Benefits:
- Flexibility to chain services hosted at multiple clouds
- Efficient cloud resource Management
- Easy to use

Operationalization of the Workflow Manager

- Integrate Workflow Manager with the AAA (Authentication, authorization, and accounting) service developed at **Deimos**
- **Collaborate with OGC Testbed 15 activities (i.e. FCA Thread) to find common data-models for Service/Application metadata**
- **The Service/Application Owners:**
  - Retaining governance over their application
  - Applying specific Access Privilege policies based on User Profile
  - Business Intelligence on the usage of their Applications
- **Users:**
  - Creating User profiles in the AAA service
  - Registering to the AAA service with the necessary keys for multi-cloud Cloud Deployment (e.g. AWS, DIAS, Azure etc.) and execution
  - Single-Sign-On enables for seamless experience
  - Receive notification during Workflow Composition of any access restrictions
  - Potentially receive resource usage statistics and billing information (multi-cloud deployment and execution)


Intelligent Identity and Access Management
AAA Service
Accounting Capabilities
Developers & Application Owners
User Authentication (profiles)
Authorization Checks
Develop & Publish
Workflow Manager
Deploy & Execute
Services and Applications
Access
Users

C.2 27 November 2019



deimos
elecnor group

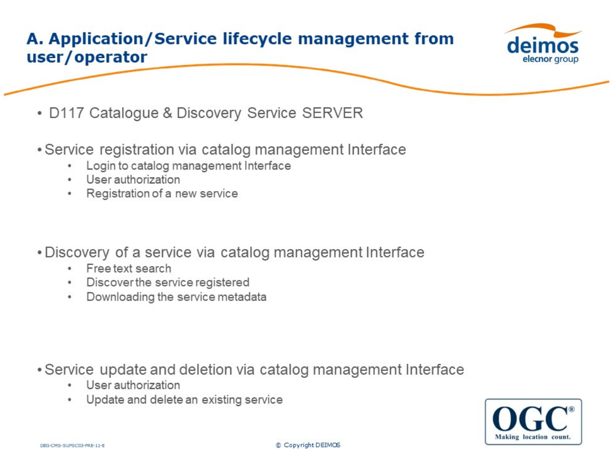
OGC Testbed 15 final demo

David Cordeiro
João Pedro Martins Serras
Koushik Panda

27-11-2019

OGC® Making location count.

DEG-CMS-SUPSC03-PRE-11-E © Copyright DEIMOS 1



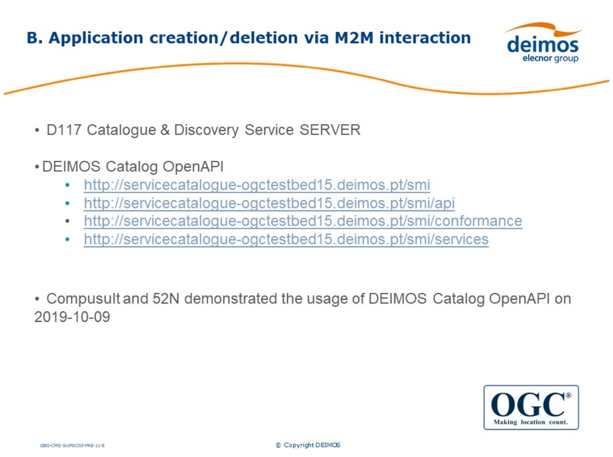
A. Application/Service lifecycle management from user/operator

deimos
elecnor group

- D117 Catalogue & Discovery Service SERVER
- Service registration via catalog management Interface
  - Login to catalog management Interface
  - User authorization
  - Registration of a new service
- Discovery of a service via catalog management Interface
  - Free text search
  - Discover the service registered
  - Downloading the service metadata
- Service update and deletion via catalog management Interface
  - User authorization
  - Update and delete an existing service

OGC® Making location count.

DEG-CMS-SUPSC03-PRE-11-E © Copyright DEIMOS



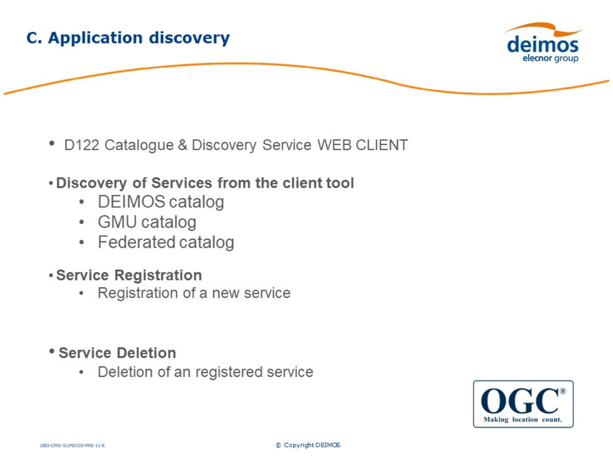
B. Application creation/deletion via M2M interaction

deimos
elecnor group

- D117 Catalogue & Discovery Service SERVER
- DEIMOS Catalog OpenAPI
  - http://servicecatalogue-ogctestbed15.deimos.pt/smi
  - http://servicecatalogue-ogctestbed15.deimos.pt/smi/api
  - http://servicecatalogue-ogctestbed15.deimos.pt/smi/conformance
  - http://servicecatalogue-ogctestbed15.deimos.pt/smi/services
- Compusult and 52N demonstrated the usage of DEIMOS Catalog OpenAPI on 2019-10-09

OGC® Making location count.

DEG-CMS-SUPSC03-PRE-11-E © Copyright DEIMOS



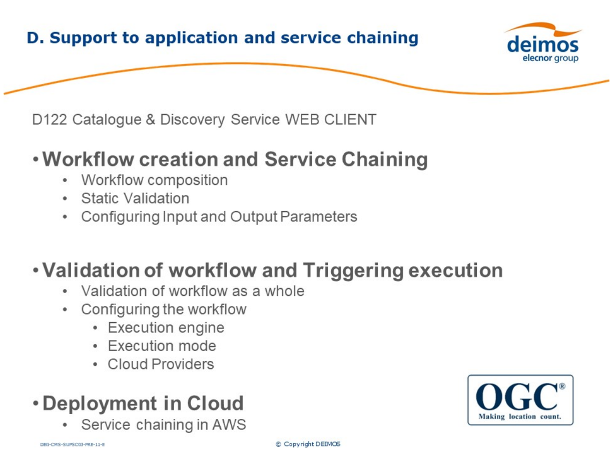
C. Application discovery

deimos
elecnor group

- D122 Catalogue & Discovery Service WEB CLIENT
- **Discovery of Services from the client tool**
  - DEIMOS catalog
  - GMU catalog
  - Federated catalog
- **Service Registration**
  - Registration of a new service
- **Service Deletion**
  - Deletion of an registered service

OGC® Making location count.

DEG-CMS-SUPSC03-PRE-11-E © Copyright DEIMOS



D. Support to application and service chaining

deimos
elecnor group

D122 Catalogue & Discovery Service WEB CLIENT

- **Workflow creation and Service Chaining**
  - Workflow composition
  - Static Validation
  - Configuring Input and Output Parameters
- **Validation of workflow and Triggering execution**
  - Validation of workflow as a whole
  - Configuring the workflow
    - Execution engine
    - Execution mode
    - Cloud Providers
- **Deployment in Cloud**
  - Service chaining in AWS

OGC® Making location count.

DEG-CMS-SUPSC03-PRE-11-E © Copyright DEIMOS

Thank you

www.elecnor-deimos.com

DEG-CMS-SUPSC03-PRE-11-E © Copyright DEIMOS