HELLENIC MEDITERRANEAN UNIVERSITY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

BACHELOR THESIS

Implementation of a federated workflow execution engine for life sciences through virtualization services.

EMMANOUIL KOUTOULAKIS

SUPERVISOR

PROFESSOR MANOLIS TSIKNAKIS

HERAKLION CRETE
AUGUST 2020

Περίληψη

Οι ροές εργασίας χρησιμοποιούνται ευρέως για την αναπαράσταση μεγάλων επιστημονικών εφαρμογών που διευκολύνουν την εκτέλεση τους σε κατανεμημένα συστήματα όπως clusters ή cloud. Ωστόσο, τα συστήματα ροής εργασίας είναι άγνωστα ως προς το περιβάλλον στο οποίο αναμένεται να εκτελεστεί κάθε βήμα της ροής εργασίας. Ως αποτέλεσμα, μια ροή εργασίας μπορεί να εκτελεστεί σωστά στο περιβάλλον στο οποίο σχεδιάστηκε, αλλά στη συνέχεια εκτελώντας την σε άλλο περιβάλλον, είναι πιθανό να αποτύχει λόγω διαφορών στο λειτουργικό σύστημα, τις εγκατεστημένες εφαρμογές, τις εκδόσεις βιβλιοθήκης, τα διαθέσιμα δεδομένα και άλλες εξαρτήσεις του περιβάλλοντος. Αυτός ο παράγοντας είναι ένα σημαντικό εμπόδιο στην βιοπληροφορική και γενικότερα στις επιστήμες δεδομένων.

Οι τεχνολογίες container όπως το Docker προέκυψαν πρόσφατα ως λύση σε αυτό το πρόβλημα παρέχοντας ένα καθορισμένο περιβάλλον εκτέλεσης σε επίπεδο λειτουργικού συστήματος. Με την χρήση αυτών των τεχνολογιών, μία πολύπλοκη ροή εργασίας μπορεί να εκτελεσθεί σε ένα απομονωμένο περιβάλλον στο οποίο ενσωματώνονται όλα τα απαραίτητα εργαλεία και βιβλιοθήκες για να πραγματοποιηθεί η εκτέλεση. Πιο συγκεκριμένα, τα Containers λειτουργούν ως ένα ελαφρύ ανεξάρτητο λειτουργικό σύστημα μέσα στο υπάρχων σύστημα το οποίο μπορούμε να το επεξεργαστούμε εύκολα και γρήγορα χωρίς τον φόβο για οποιαδήποτε καταστροφή στο σύστημα φιλοξενίας του.

Η ευελιξία και η φορητότητα είναι σημαντικά προβλήματα στην εκτέλεσης μίας επιστημονικής ροής εργασίας. Υπάρχουν πολλές αξιόλογες πλατφόρμες που σχεδιάζουν και εκτελούν αυτές τις ροές. Κάθε μία εξ ’αυτών έχει την δική της γλώσσα-τρόπο για την αναπαράσταση μιας ροής εργασίας. Ως εκ τούτου, μία ροή εργασίας η οποία έχει αναπαραχθεί σε μία συγκεκριμένη πλατφόρμα εκτέλεσης δεν μπορεί να εκτελεσθεί σε διαφορετική πλατφόρμα λόγο της διαφορετικής γλώσσας η οποία χρησιμοποιήθηκε για την παραγωγή της. Κύριο αποτέλεσμα αυτού, είναι να απαιτείται από τον εκάστοτε χρήστη, εξειδικευμένη γνώση για την ανάπτυξη και την επεξεργασία μιας ροής εργασίας. Επιπλέον, οι πλατφόρμες αυτές δεν είναι συνδεδεμένες με κάποιο αποθετήριο έτσι ώστε να γίνετε άμεσα η εκτέλεση και η εξαγωγή αποτελεσμάτων μία ροής εργασίας με αυτού, το οποίο είναι ακόμα ένα μείζον πρόβλημα για την διαλειτουργικότητα μιας επιστημονικής ροή εργασίας.

Το OpenBio (https://www.openbio.eu) είναι μια διαδικτυακή πλατφόρμα η οποία είναι υπό ανάπτυξη από το Εργαστήριο Υπολογιστικής Βιοϊατρικής του Ινστιτούτου Πληροφορικής του Ίδρυματος Τεχνολογίας και Έρευνας σε συνεργασία με το Πανεπιστήμιο Πάτρας για την κατασκευή και αποθήκευση ροών εργασίας που μπορούν να συνθέσουν πολλά εργαλεία ή ροές εργασίας. Αυτή η πλατφόρμα έχει ως στόχο την μεγιστοποίηση της αναπαραγωγιμότητας και την ενοποίηση κοινωτήτων της Βιοπληροφορικής. Η ροή εργασίας αναπτύσσετε χρησιμοποιώντας BASH εντολές η οποίες είναι λειτουργικές είτε σε τεχνολογίες container ή σε περιβάλλοντα εκτέλεσης ροών εργασίας.

Στην εν λόγω πτυχιακή εργασία, εξετάσαμε τον τρόπο με τον οποίο μπορούμε να ενσωματώσουμε στην πλατφόρμα OpenBio ένα περιβάλλον εκτέλεσης που εκτελείται σε εικονικό container μέσω ενός τοπικού υπολογιστή, cluster ή cloud. Με αυτήν την λειτουργία, οι χρήστες μπορούν να διαχειριστούν πολλές ροές εργασίας, να παρακολουθήσουν τη χρήση των πόρων οι

οποίοι καταναλώνονται από το σύστημα κατά την διάρκεια μίας εκτέλεση, οι οποίες μπορούν να συμβάλλουν στην επίτευξη κλιμάκωσης ή βελτιστοποίησης χρήσης πόρων.

Στα πλαίσια της εργασίας αυτής, δοκιμάσαμε διάφορα συστήματα διαχείρισης ροών εργασίας και εργαλείων παρακολούθησης πόρων. Αυτή η προσεκτική αξιολόγηση είχε ως αποτέλεσμα την ακόλουθη σειρά εργαλείων. Το AirFlow το οποίο είναι ο μηχανισμός για την εκτέλεση των ροών εργασίας(Workflow Management System), το NetData για την παρακολούθηση πόρων και ένας διακομιστής που γράφτηκε σε Python Flask και ενεργεί ως API για την επικοινωνία με την πλατφόρμα OpenBio. Όσον αφορά τον μηχανισμό εκτέλεσης, το Airflow δεν είναι το μόνο εργαλείο που θα μπορεί να ενσωματωθεί μέσα στο περιβάλλον, καθώς σκοπός είναι το σύστημα είναι συμβατό με πολλαπλούς μηχανισμούς εκτέλεσης ροών εργασίας. Επίσης, χρησιμοποιήθηκε το Docker-Compose για οργάνωση των containers για την καλύτερη επικοινωνία και διαλειτουργικότητα μεταξύ των εργαλείων αυτών. Η πλατφόρμα OpenBio σε συνεργασία με το περιβάλλον εκτέλεσης παρέχει:

(1) μια διεπαφή χρήστη(User Interface) που επιτρέπει στους χρήστες την δημιουργία, επεξεργασία, επεκτασιμότητα και αποθήκευση σύνθετων ροών εργασίας χωρίς την ανάγκη επιπλέον γνώσεων προγραμματισμού (παρά μόνο τις απαραίτητες γνώσεις BASH scripting) χρησιμοποιώντας την πλατφόρμα OpenBio,

(2) φιλικό προς το χρήστη σύστημα παρακολούθησης πόρων σε πραγματικό χρόνο,

(3) αυτόματη δημιουργία αναφορών με αποτελέσματα και αρχείων καταγραφής κατά την διάρκεια της εκτέλεσης,

(4) φορητότητα σε κατανεμημένα υπολογιστικά περιβάλλοντα όπως clusters και clouds με δυνατότητα δημιουργίας πολλαπλών παρουσιών με την χρήση των containers.

Στην παρούσα πτυχιακή εργασία, αναγράφονται τα εργαλεία τα οποία χρησιμοποιήθηκαν για την ανάπτυξη του περιβάλλοντος εκτέλεσης, η αρχιτεκτονική αυτού, αναλυτικές οδηγίες για την εγκατάσταση του περιβάλλοντος εκτέλεσης σε οποιοδήποτε σύστημα και ένα παράδειγμα εκτέλεσης μία επιστημονικής ουράς εργασίας με μετρήσεις εκτέλεσης ενός παραδείγματος ως προς τον χρόνο και σύγκριση με άλλους τρόπους εκτέλεσης.

Συνολικά, το περιβάλλον εκτέλεσης το οποίο αναπτύχθηκε για να ενσωματοθεί στην πλατφόρμα OpenBio για να διευκολύνει τους χρήστες στην εκτέλεση πολύπλοκών επιστημονικών ροών εργασίας. Ωστόσω, ο τρόπος ο οποίος αναπτύχθηκε όλο το σύστημα μπόρει εύκολα να ενσωματωθεί σε οποιαδήποτε πλατφόρμα με τις κατάλληλες παραμετροποιήσεις.

Λέξεις-Κλειδιά

workflow, container, Docker, Airflow, NetData, OpenBio, bioinformatics, Executional Environment, reproducibility, portability.

Abstract

Workflows are widely used abstractions for the representation of large scientific applications that also ease their execution on distributed systems such as clusters, clouds, and grids. However, workflow systems are mainly agnostic on the environment on which each task of the workflow is expected to run. As a result, a workflow may run correctly in the environment in which it was designed, but then moved to another environment, it is likely to fail due to differences in the operating systems, installed applications, library versions, available data, and other dependencies. This factor is a major issue in life sciences. Lightweight container technologies like Docker have recently arisen as a solution to this problem by providing a well-defined execution environment at the operating system level.

OpenBio (https://www.openbio.eu) is a web-based workflow platform that can compose multiple tools or workflows in one and aims to maximize reproducibility. In this thesis, we consider how to best integrate the OpenBio platform with an Execution Environment running in a virtual container. With this abstraction, users can manage multiple workflows, monitor the use of their resources, which can help achieve scalability and optimal resource utilization. Several platforms currently exist that design and execute sophisticated pipelines (e.g Galaxy [7], Luigi [8], Nextflow [9]). The main drawback of these platforms is the lack of the necessary parallelism, flexibility, and portability.

In this thesis, we test a variety of workflow management systems and resource monitoring tools. This careful evaluation resulted in the following stack of tools. AirFlow is used for Workflow Execution, NetData used for resource monitoring, and a client written that uses Python Flask acts as an API for interface monitoring. Also, we leverage Docker-Compose to orchestrate the communication and interoperability between these tools. AirFlow was used due to its ability to treat scientific pipelines in a simple, portable, reproducible, and scalable manner, mainly by modeling them as DAGs (Directed Acyclic Graphs). AirFlow and NetData both are configured in accordance with the Execution Environment prerequisites. OpenBio Platform with the collaboration of the Execution Environment provides; (1) a drag and drop user interface using OpenBio platform for pipeline composition that allows users to create complex pipelines without familiarity in underlying programming languages, (2) User-friendly monitoring system, (3) automatic report generation with results and processing logs and (4) portability towards distributed computing environments such as cluster, grid, and cloud with the ability to generate multiple instances.

Keywords: workflow, container, Docker, Airflow, NetData, OpenBio, bioinformatics, Executional Environment, reproducibility, portability.

Πρόλογος

Στο σημείο αυτό θα ήθελα να ευχαριστήσω τους ανθρώπους που έπαιξαν καθοριστικό ρόλο στην εκπόνηση της πτυχιακής μου, αλλά και ευρύτερα στην πορεία μου σε ακαδημαϊκό επίπεδο.

Αρχικά, θα ήθελα να ευχαριστήσω, τον επιβλέποντα καθηγητή της πτυχιακής μου, κύριο Μανόλη Τσικνάκη, Καθηγητή του τμήματος Μηχανικών Πληροφορικής του Ελληνικού Μεσογειακού Πανεπιστημίου, για την συνεργασία μας, καθώς και για τον πολύτιμο χρόνο που μου αφιέρωσε.

Στην συνέχεια, θα ήθελα να εκφράσω την ιδιαίτερη ευγνωμοσύνη μου προς τον κύριο Λευτέρη Κουμάκη, Καθηγητή του τμήματος Μηχανικών Πληροφορικής του Ελληνικού Μεσογειακού Πανεπιστημίου για τις συνεχείς συμβουλές του, τόσο στο θεωρητικό κομμάτι όσο και στο πρακτικό του ενδιαφέροντος μου για ποικίλες πτυχές των υπολογιστικών συστημάτων, καθώς και στην μορφοποίηση της εργασίας.

Οφείλω ένα ακόμα μεγάλο ευχαριστώ, στον κύριο Αλέξανδρο Καντεράκη, Ερευνητή στο Εργαστήριο Υπολογιστικής Βιο-Ιατρικής του Ινστιτούτου Πληροφορικής στο ΙΤΕ, τόσο την συνεργασία μας, όσο και στην διαμόρφωση του τρόπου σκέψης μου ως προς την προσέγγιση διαφόρων ζητημάτων κατά την διάρκεια της εκπόνησης της πτυχιακής μου, τον χρόνο και τις γνώσεις που αφιέρωσε ώστε να ανταλλάξουμε σκέψεις και ιδέες σχετικά με σημαντικά ζητήματα που προέκυψαν κατά την ανάπτυξη του περιβάλλοντος εκτέλεσης.

Τέλος, θα ήθελα να ευχαριστήσω εκ βάθος καρδίας τους γονείς μου για την αγάπη και τη στήριξη που μου προσφέραν κατά την διάρκεια της εκπόνησης της πτυχιακής μου εργασίας.

Μάνος Κουτουλάκης,

Ιούλιος 2020.

Contents

List of Images

List of Tables

1 Introduction

In this chapter, we describe the background on Workflow Management Systems in bioinformatics and present some gaps in existing systems. These gaps, along with their dire consequences in reproducibility, motivated most of this work in this thesis. Furthermore, we present the general principle of my implementation and a short discussion on the scientific merits of this effort.

1.1 Background

Efficient and cost-effective analysis of high-throughput data is now broadly considered a major bottleneck in bioinformatics [1]. As a result, the optimal utilization of computation resources is a far more important factor than computational power. Consequently, we have important impacts on budgetary decisions [2]. The most significant complexity of high-throughput sequencing data analysis is that a tremendous number of different steps are frequently executed with a set of programs with different interfaces, dependencies and architectures. Thus, each sequence analysis requires the integration of components made with different programming languages and computation setups. Here we argue that this complexity can be significantly reduced by applying component isolation. This isolation is easily achieved today with special tools that offer "*virtualization*". Virtualization software encapsulates a set of tools, services and configuration scripts along with the underlying operating system in an isolated component, called "container". Containers act as independent software, they can run concurrently on the same physical server, they can communicate and they can be stopped and started at will. Also, containers can just be copied and deployed in multiple execution environments, simplifying the process of scaling a demanding computation procedure. All of the above, contribute to the reduction of the complexity which leads to explicit cost reduction in service and maintenance cost [3]. One of the most known virtualization software is Docker. Docker uses a containerization technique that has increased popularity and has brought forward the term "*container management software*" [4]. Furthermore, Docker-Compose [5], a tool of Docker that is responsible for orchestrating containers, as it executes multi-container Docker applications. Therefore, the usage of docker is important for this dissertation since it can isolate the platform from the host. Also, the container is very lightweight and easy to be handled such as to create, edit or remove it from the hosting service [6].

1.2 Previous Research

Some scientific workflow management systems such as Galaxy [7], Luigi [8] and Nextflow [9] have implemented workflow management. All of them have the same actions as unloading and executing complicated workflows using a specific workflow language. Also, platforms such as Galaxy and Nextflow have already supported Docker and that give us the opportunity to solve portability problems. Despite the fact that such platforms are a robust way to merge existing programs into pipelines that carry end-to-end data processing, they are restricted in their flexibility. In other words, all of these systems have different workflow file types and cannot cooperate. As a result, researchers are stacked at one workflow management system and they cannot use their existing pipelines/workflows on other management systems. In addition, many of these workflow management systems lack parallelization execution and the latency of workflow is increased

according to the number of steps of execution. Also, it cannot be denied that no one from these Workflow Management Systems have resource monitoring which is essential for users to monitor a workflow during the workflow execution. For this reason, flexibility, parallelism and resource monitoring have become necessary not only for bioinformatics workflows but generally in scientific workflows to become more creative and easier to resolve errors.

1.3 Problem Formulation

Flexibility is an integral problem of workflow execution. Nowadays, many different workflow management systems were introduced in the research community, each of them having its own workflow language or library to be executed. Consequently, none of these can be used on another WMS except its own. As a result, researchers are bewildered about which of these WMS is better for their specific task, creating a vicious circle in the research community, thus, increasing the complexity of the workflow execution. During these selfish innovations, we have forgotten the simplicity and significance of the pure BASH scripting. To address the compatibility of workflows we already have integrated a variety of workflow management systems in our environment. This feature can make it easy for users to run any type of workflow they want in our OpenBio environment according to their workflow type. About the BASH scripting, the OpenBio can parse a workflow which is written in BASH at any workflow language according to the workflow executor.

The computing environments have grown in complexity, which is another negative aspect, thus frustrating researchers on how to handle and integrate their WMS. To clarify, every educational and/or research institution depends on various sets of computing options such as servers, computing clusters, and cloud computing. Consequently, the Execution Environment should be portable and utilizable in different environments. Hence, it should not be installable only on central computer but should also create many isolated executable environments on the same computer in case the system has more than one user.

The Execution Environment should be able to run large workflows without conflicts, and in the event of a collision, the Environment would auto recover from any problems. To put it briefly, the virtualization platform should provide self-healing that automatically diagnoses and repairs software problems. Additionally, since the Environment executes complex pipelines, it requires a resource monitoring system to monitor the resources used during the execution of a workflow. Many of common WMSs have not any resource monitoring provider to monitor their workflows.

1.4 Motivation

We believe that combining both the required tools using containerization could allow better isolation and performance for our execution environment. The containers of the Execution Environment will be federated but diversified from the host Operating System. Taking into consideration the advantage of containerization and the orchestration, this project was built using both Docker and Docker Compose, which make the execution environment portable. Since Cloud Computing raised rapidly and made containerization technology a significant factor in scientific society, every one of us prefers to deploy a WMS into a cloud to make the execution of a workflow. Hence, the usage of docker is the more compatible solution in our project.

It could also be argued that many workflows are not being able to embed and to be embedded. To clarify, there are a plethora of Workflow Management Platforms that offer different libraries, and programming languages. As a consequence, one workflow can be executed in specific WMS. In this thesis, we took into consideration the architecture of the execution environment to be modular and set the client to act as a workflow parser in place. That is to say, the client is easy to allow other workflow execution engines as well. Also, it cannot be denied that workflow executors should be editable and be able to run steps in parallel. Another important feature is the resource monitoring system that is integrated into the execution environment and users can receive extensive resource information about the process of performing their workflows.

1.5 Scope

In this dissertation, we proposed to utilize a group of containers as the execution engine for scientific workflows, having in mind the current limitations of the existing systems. First and foremost, it perfectly solves the scientific tool installation problem. We packaged scientific tools into the OpenBio Platform, and we can set up them as steps into the pipeline saved on the OpenBio server. Subsequently, on the OpenBio platform, we give the environment variables that are needed for the execution and instantly we can send it in our execution environment as a parsed file according to which WMS (Workflow Management System) is in use. Secondly, this execution engine is portable and utilizable, because of the container-based architecture. In other words, this workflow engine could work on any hosts only by using the docker-compose.yml file that contains the WMS, database, and resource monitoring system. Third, the engine is absolutely isolated. Everyone can have multiple execution engines on a computer cluster or at the same host. Fourth, the user can integrate any WMS at the execution environment such as AirFlow, NextFlow, etc. As we mentioned above, the WMS that we test is Airflow is not bound in our Execution Environment. Finally, the same goes for the OpenBio platform, it could also work perfectly with other workflows platforms by virtue of handling API easily.

1.6 Target

The target of this thesis is to resolve workflow portability problems as well as collaboration with other workflow management systems to make the execution simple and beneficial. Previously, we referred to the OpenBio platform which is the main reason that this engine was built and consequently integrated into it. Since our testbed is the OpenBio platform, for the first step the user should make an account into the system. Then, could trigger the workflow and download the results of the logs. In parallel, users have the opportunity to see the resource monitoring of their execution engine. Below we provide an extensive report on the tools used to implement the mechanism. All these will be able to centralize the bioinformatic field into one repository which can execute, share, and publish the results of scientific research. The right usage of the execution environment could facilitate scientific research by providing a variety of WMS without the need for additional knowledge of workflow language except for the BASH script.

1.7 Thesis Structure

This chapter outlines the different sections of the project report.

- **Chapter 2**: *Technical Background* that provides the background information about virtualization and generally for technologies used to build the workflow execution environment.

- **Chapter 3:** System *Design. This chapter* refers to the architecture of the thesis, containing code and design decisions.

- **Chapter 4**: *Implementation*. That section contains a demonstration of the central points of my development process. Also, include main details about the implementation of the execution environment abilities.

- **Chapter 5**: *Integration & Experimental Results.* In this section, I refer to the integration of the Execution Environment. Furthermore, we introduce an example by running a workflow in the Execution Environment in collaboration with OpenBio.

- **Chapter 6**: *Conclusion.* Concluding remarks and future improvements and extensions to my project.

2 Technical Background

In this chapter, we describe useful tools that contribute to finalizing the Execution Environment. Every tool below is described according to which is the usage and what facilitates. Moreover, we present some figures to be more comprehensive in the usage of them.

2.1 Virtualization Technology

Virtualization technology is mentioned as the abstraction of computing resources such as memory, storage, CPU, database from applications and end users consuming the service. Virtualization technology counts on software components to simulate the hardware functionality by creating virtual resources. The main motivations of virtualization are isolation and rapid elasticity. More concretely, with virtualized environments, two or more customers can co-exist on the same host without interference [28]. Every one of these environments is limited to its own context and will not be aware of other environments unless specifically defined on the host. Nowadays, virtualization is used at hardware and operating system level.

2.1.1 Operating System Level Virtualization

Operating System Level Virtualization has acquired traction over the years. Hardware level virtualization is considered as heavyweights because it relies on hardware emulation. Otherwise, there is containerization that uses kernel features like cgroups (control groups), namespaces etc., creating isolated instances known as containers, on the top of the host machine as it depicted in **Fig. 1**. More specifically, the containers share the host machine's kernel with the help of the container engine rather than running a full operating system. Consequently, containerization technology reduces the overall overhead.

Containerization was developed in the UNIX operating system back in 1979 using chroot [30]. Subsequently, as containerization technology evolved, more essential features were implemented for file system, users, and networking isolation. The first container manager is LXC (Linux Containers), then Docker is represented with a full ecosystem to manage containers.

2.1.2 Hypervisor-based virtualization

On the other hand, a Virtual Machine (VM) is a simulated machine that runs into another physical or virtual machine. In the way of physical machines, VM acts the same, but it has emulated hardware. The machine that hosts the VM is named Hypervisor. The hypervisor can manage multiple VMs, which signifies that more than one user can effectively be isolated and concurrently served within a single physical machine. A VM runs on its own Operating System that does not integrally have to be the same as the host machine. All of the VMs are found on disk images which are either operating systems or packaged together with software. Also, VM is utilizable and it can be paused or stopped and its state can be saved to a new image.

Fig. 1 shows us the differences between Virtual Machine and Containerization technology.

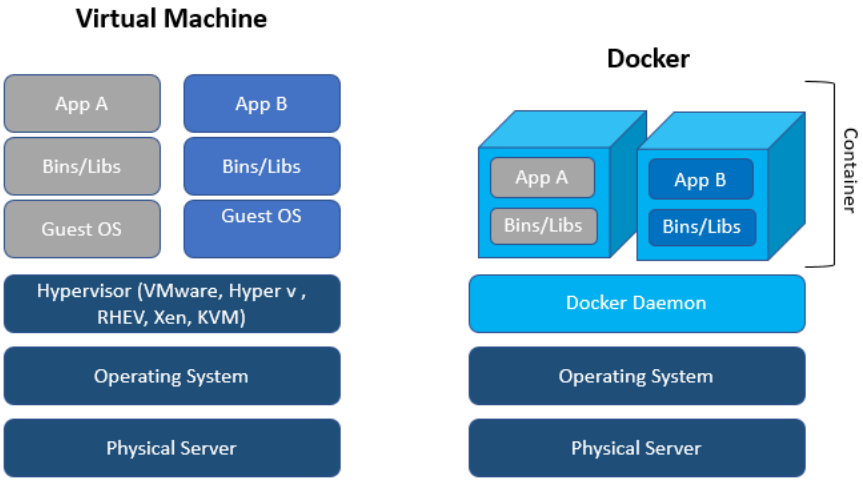

Figure 1. Comparing Virtual Machines (VM) Docker containerization Technology

2.2 Container-based Virtualization

As we mentioned before, an application container is an isolated unit of software that is packaged code so it can be run dependably from all computing environments. Furthermore, a container image is lightweight because only the dependencies of software are installed inside of the container. Consequently, containerization methods make the application integration simpler and applicable to all data centers, public clouds, or even a developer's computer [13]. The most common containerization solution is Docker, but Singularity [24], Shifter [25] are recent alternatives that prevent users from running containers with root privileges, addressing most common security issues when deploying containers in multi-tenant computing clusters such as on high-performance computing (HPC) clusters. Docker containers are usually shared via Docker Hub (https://hub.docker.com/), but there are also initiatives for standardizing containers in the life sciences like BioContainers [26]. As we mentioned above, the containers are stand-alone and that is the reason that we decided to integrate the execution environment into containers. **Fig. 1** shows us a diagram that explains the container-based virtualization and what are the differences between containers and VMs.

2.2.1 Docker

To be able to facilitate the execution environment to be isolated and portable, software for management and runtime is required. Docker is practical to use because it has a large active community and is rapidly growing in popularity among the bioinformatics. Another great aspect of Docker is that the containers are system-agnostic by doing them isolated from the host's OS [14]. More specifically, Docker uses images as a basis for the container's creation. Also, users can set environment variables or add software with complex dependencies via Dockerfile. Then, Docker builds the container based on that Dockerfile, creating an executable package with the

dependencies set. from the user. Docker adds a layer on top of the host OS for controlling the containers during the build process and when the container runs. Generally speaking, Docker solves the issues of portability and consistency between environments. Portability in Docker is not represented by the possibility to migrate VMs or OSs but it makes it possible to ship only the code of the application.

2.2.2 Docker Compose

Docker-Compose is a Docker tool that is utilized to run isolated environments as containers that build and run an application. Docker-compose simplifies the process of setting up and running the applications by defining a YAML file to configure your application's services. Then, we can create and run all the federated services that contain in a YAML file with a single command. The YAML is a format to create human-readable files and a great tool to construct a configuration file. Undoubtedly, Docker Compose facilitates the integration of the Execution Environment for any cloud provider, personal computer, or cluster. Table 1 is an example of docker-compose.yml that is for container orchestration.

```
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
    environment:
      FLASK_ENV: development
  redis:
    image: "redis:alpine"
```

Table 1. Example of docker-compose.yml file.

2.3 Web Application

In general, Web application is a client-server system where a browser represents the client and a web-server as the server. Web application logic is the relation between the client and the server, data storage is performed mainly on the server. Data is interchanged over the network through the Hypertext Transfer Protocol (HTTP). This approach takes advantage of the web and is the fact that users do not depend on a specific operating system or hardware configuration. Thus, web applications are cross-platform services and provide interoperability due to the containerization techniques which can be used for the development.

2.3.1 Flask Framework

Flask [21] is a python-based framework for Web-Applications. This framework supports extensions that can add application capabilities as if they were applied to Flask itself. There are several extensions such as validation form, upload handling, and generally several common framework related tools. These extensions used to be updated more often than the core Flask framework. The main components of the Flask are:

- **Werkzeug:** It is a toolkit for Web Server Gateway Interface (WSGI) application. Werkzeug can perform software objects for request, response, and utility functions.
- **Jinja Template:** It is a template engine for the Python programming language that handles templates in sandbox. Jinja has an expressive language that gives template authors a more robust set of tools.

Our Execution Environment is implemented with the Flask framework. The system provides a useful API for create, update and delete workflows from the environment. Additionally, facilitates communication using requests to handle and get information from the other tools that contribute to our environment.

2.4 Back-end

Back-end development is the implementation of server-side, which focuses on web application logic or, in other words, how the application works. It is a process of creating the core of a web application, developing the platform for the application and filling it with all the required functionality. The Server-side manipulates the data that is received from the front-end and returns the results back in the form that is understandable by the client-side. For our circumstances, the Back-end was implemented using a Python framework and it was necessary for the API implementation and for the communication with the OpenBio platform. Back-end usually comprises three parts: a web server software, an application logic, and a database.

2.5 Workflow Management System

Workflow Management System (WMS) plays an important role for scientific computing. WMS is designed to compose, edit, share and execute a sequence of computational steps, or workflows in a scientific application. The purpose of WMS is the automation of complex processes on large volumes of data, becoming more agile, reducing costs, and increasing productivity. In addition, WMS can visualize workflows using diagrams, depicting inputs, outputs of workflow, and allow to save workflow for sharing and publishing [10].

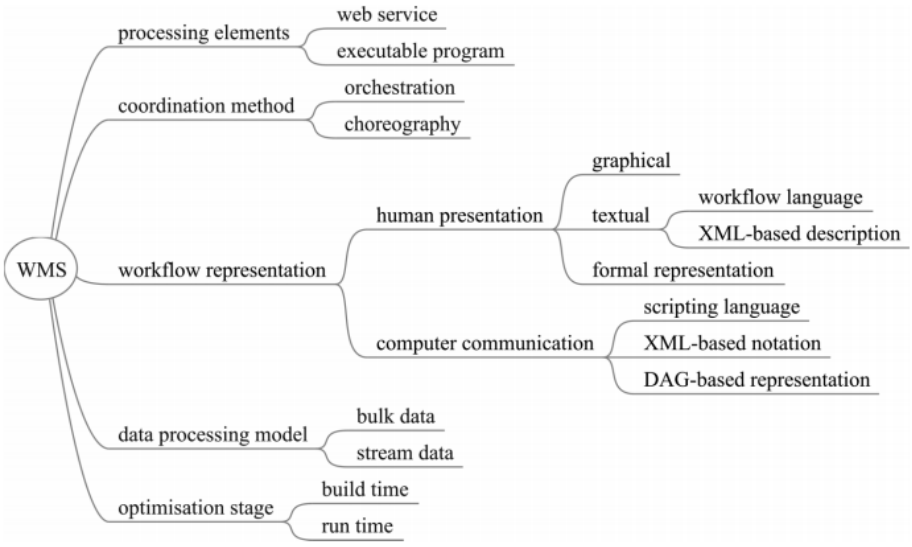

Figure 2. Architectural characterization of WMSs

Many scientific WMSs emerged with the diffusion of Cloud Computing, Web Service, and Grid technologies, which offered the possibility to access robust services and infrastructures in a more natural way than before [11]. Therefore, they were mainly targeted towards these architectures and not focused on portability. Nevertheless, by evolving in strict contact with the scientific community, they acquired maturity from the functional design point of view and established consensus among researchers. Moreover, some of them currently provide workflows repositories or are evolving to support diverse newer architectures. Some well-known WMS are Galaxy [7], Apache Taverna [12] that includes an interface allowing users to build and modify complex workflows with little to no programming knowledge. Thanks to these systems, researchers are able to focus on their research issues rather than worrying about the workflow execution mechanism.

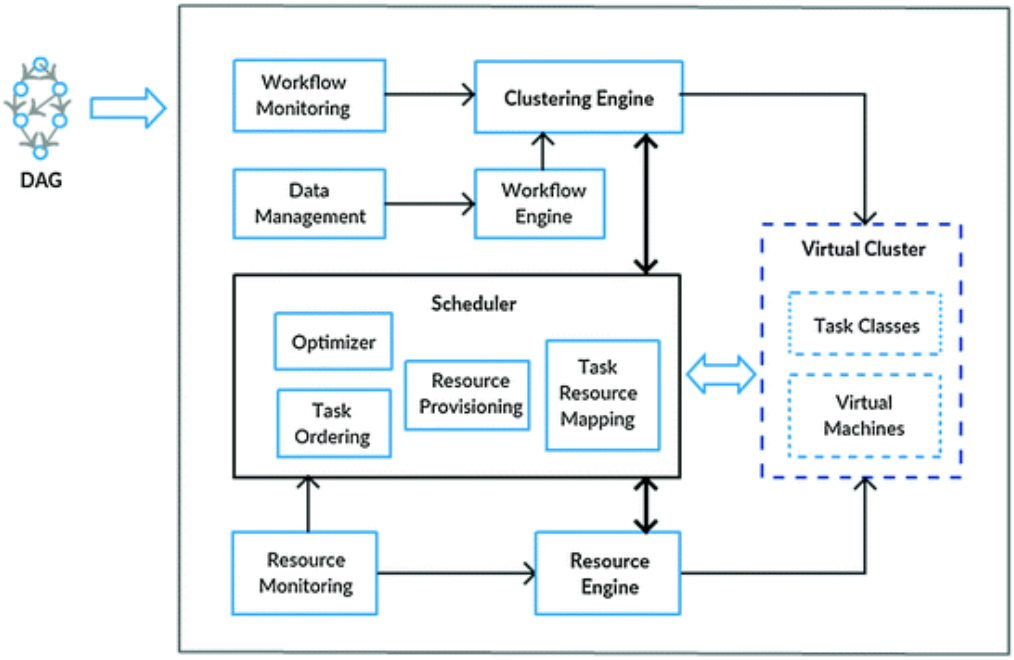

Figure 3. Architecture for Scientific Workflow Management.

Taking into consideration how Big Data are spreading in every scientific field, dataflow management is growing. As a result, more and more workflow languages, libraries and systems arise, and that restricts the research. For this reason, the execution environment can be compatible with many Workflow management systems. Below, we refer to some of these systems but by the time only the Airflow [15] is integrated. To clarify, Airflow is not the only solution, there are many WMS that could work in our Execution Environment without execution problems.

- **Airflow** [15] is a lightweight workflow manager. Developed by Airbnb, it is now maintained by Apache Incubator. Airflow executes workflows as directed acyclic graphs (DAGs) of tasks. Every task is standalone and does not share any resources with other tasks. The DAG objects are utilized from Python scripts describing the relationship between the tasks and their order of execution. Airflow has a modular architecture and can allocate tasks to an arbitrary number of workers and across multiple servers, according to the task sequence and dependencies defined in the DAG. Airflow is easy to install, and can be used to run task-based workflows in various environments ranging from personal computers and servers to cloud environments.

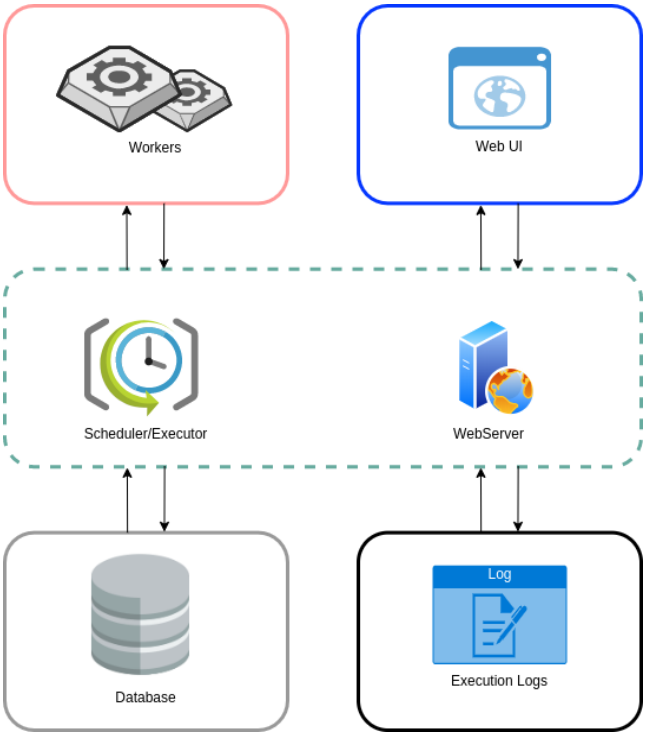

Figure 4. Apache Airflow General Architecture

- **Nextflow** [9] is developed in Java and it is a main framework for the Bioportainer Pipeline Runner based on the dataflow programming model and based on the UNIX pipe concept. Nextflow can leverage parallel execution, error tolerance, execution provenance and traceability. Parallelization, is defined by the processes inputs and outputs declarations and can scale-up and scale-out, transparently, without having a specific platform architecture. Also, this WMS works in all infrastructures as well as cloud, Docker, and Singularity. During the pipeline execution, all the intermediate results are automatically tracked. This feature allows us to resume the execution, from the last successful executed stem, no matter the reason for it stopping.

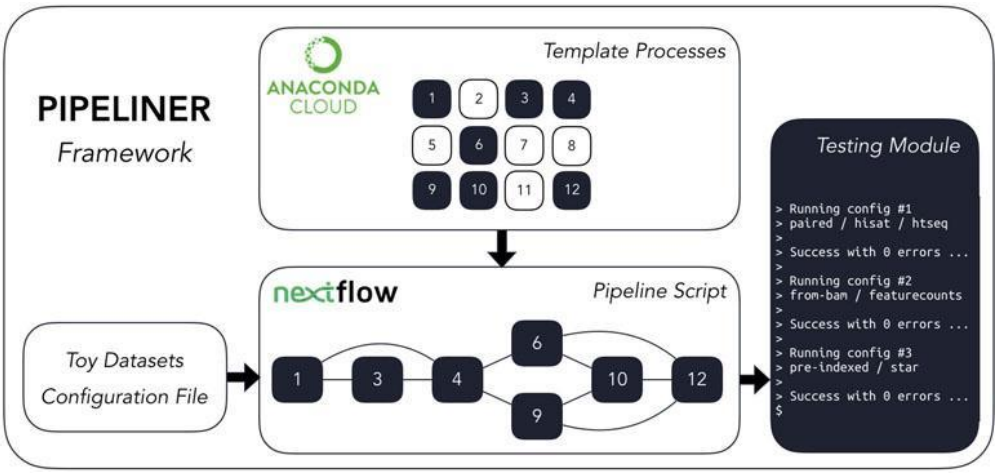

Figure 5. The Pipeliner (Nextflow-Based) framework (source: https://www.frontiersin.org/articles/10.3389/fgene.2019.00614/full)

- **Luigi** [8] is an open-source project from Spotify. It can be able to build and execute complex workflows. As I mentioned for the previous WMSs, Luigi can specify workflows as tasks and dependencies between them. Also, Luigi has a robust python package to build and run pipelines. Also, it has support for the Apache Hadoop [17] and Apache Spark [18] execution environments together with support for the local file system in the same framework. Some of the important features it provides are Workflow definition, Failure handling, Common event handling, Task tracking, Smooth integration of regular tasks and Spark jobs.

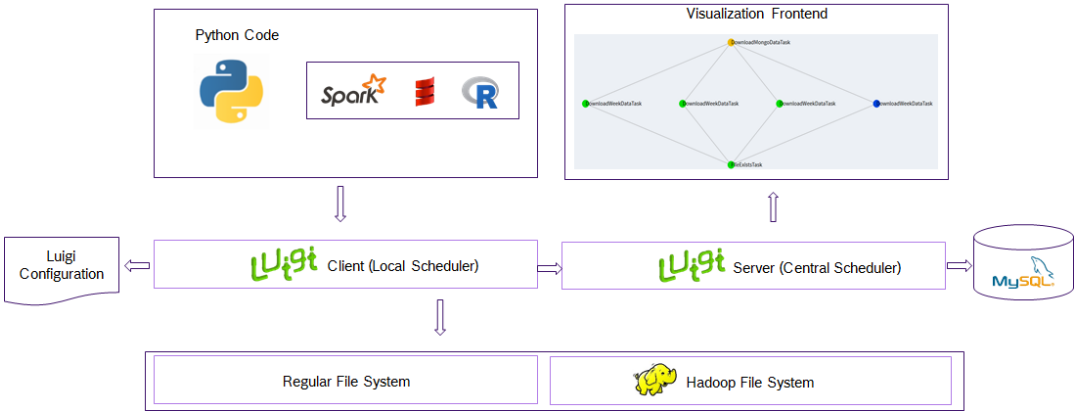

Figure 6. Luigi workflow execution diagram. (source: https://medium.com/@prasanth_lade/luigi-all-you-need-to-know-f1bc157b20ed)

2.5.1 Scientific Workflows

Generally, a scientific workflow contains isolated data transformations, analysis steps, and mechanisms to link them according to data dependencies among them. In other words, it can be represented as a sequence of computational operations or data manipulation steps to complete a process. In Bioinformatics, there are some common Workflow Management Systems like Galaxy [7], Nextflow [9], Snakemake [19] that are able to make this abstraction. Nevertheless, the flow-centric construction of workflows has been implemented from industrial design systems and is not absolutely suited to the flexibility of modern scientific research such as bioinformatics research. Consequently, the construction of workflows necessitates exquisite IT skills. In cooperation with

OpenBio.eu, users construct Workflows by simply importing bash commands that execute a step. Obviously, these commands are the same as those that they use in a terminal. According to the above, a scientific workflow emerged from the need to model complex, distributed applications. In literature, a scientific workflow is usually represented as a directed acyclic graph (DAG) [29], where nodes denote data processing tasks and the edges represent data flow. **Fig. 7** represents an example that uses BeCAS [22] to annotate NCBI Disease Corpus [23]. According to this diagram, every circle is bash commands that have to be executed and the cubes are tools that are used in the execution. During the build process, we tested many workflows to check the consistency of the execution environment.

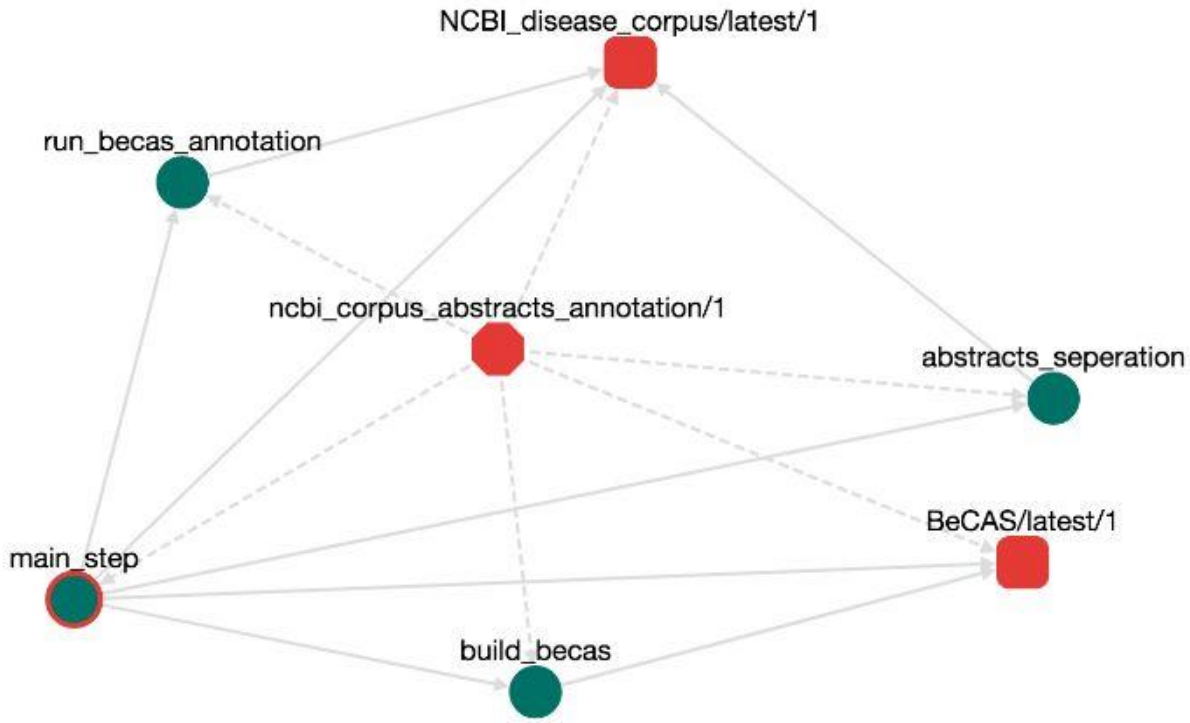

Figure 7. Workflow example in bioinformatics from OpenBio Platform

2.6 Resource Monitoring System

Resource Monitor, is a software or a service that displays information about the hardware usage throughout the system's processes. A resource monitor software includes many information about the system such as CPU, memory, disk, network etc. Modern resource monitor software has implemented more features describing and other information about the container's lifecycle and resource consumption.

During the workflow execution, the Execution Environment has an additional feature that stands to monitor the system's resource consumption. More specifically, the Environment provides extensive information about the responsiveness of the environment as well as the difficulties during the execution of a workflow.

Netdata [16] is an isolated, free, open-source, real-time performance monitoring system hosted by Cloud Native Computing Foundation (CNCF). It runs on all systems (physical and virtual servers, containers) without disrupting their core function. Also, provides a database that stores long-term resource metrics, all at 1-second, as well as could be integrated with other toolchains (Prometheus, Grafana, InfluxDB, and more). The metrics visualizer is interactive, super fast, and easy exported to a custom dashboard. This monitoring system has integrated with our execution environment that offers to user's real time information about the Disk, RAM, CPU that consumes the Execution Environment.

3 System Design

This chapter is vital to the comprehension of both the purpose and the rationale of the thesis. In this section, we make an extensive explanation of the flow and the reasoning that resulted in the implementation of features. Previously, we introduced the system that consists of different components, which will be integrated in a topology and orchestration ecosystem.

3.1 Architecture

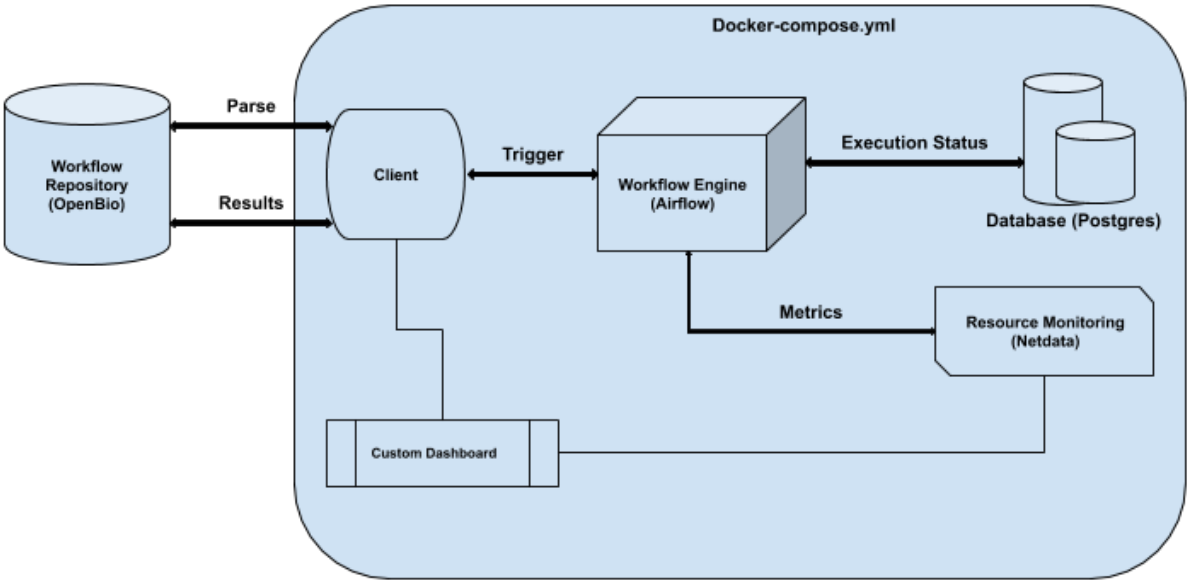

Figure 8. Execution Environment Architecture

The architecture of our system is depicted in Fig 8. It consists of 5 main components: The Workflow Repository, the Client, the Workflow Engine, the Database, and the Resource Monitoring Service. All these components are configured by using a docker-compose file. Every one of them is an isolated container except for the OpenBio server. For our case, OpenBio works as a Workflow Repository that provides a workflow to be executed. The purpose of the Client is to operate as a mediator between the Workflow repository and the Execution Environment. In addition, during the communication with the workflow repository, the workflow is parsed in a specific workflow file type (DAG) to be congruent with the Workflow Engine. Then, using the Client's API, users can edit, delete or run the workflow. The Workflow Engine (described in detail in Section 2.5) is the software that executes the workflow. When the execution finishes the reports, the logs, and the data that is used for the execution are placed into persistent volumes and are shareable into the OpenBio. Moreover, the usage of a Database is necessary to update the execution statuses. The choice of Database has to be compatible with the Workflow engine and specifically configured because of the vulnerable data that are recorded. For example, we use Airflow WMS and according to the workflow schedule used we need to integrate the PostgreSQL database. Finally, the Resource Monitoring Service (described in detail in Section 2.6) is the component that monitors the whole Execution environment, generating a custom dashboard with essential metrics such as CPU usage, RAM usage, Disk usages, and network usage. To prevent traceability, we also integrate Nginx that reverses the proxy of the Monitoring service. When the Execution

Environment starts, the custom dashboard is available in a specific URL. Aforementioned, this architecture is the same for all infrastructures such as cloud, cluster, personal computer etc.

3.2 Containers

The general benefits of containerization have already been covered in *Chapter 2*. During the years the increased popularity of Docker special attention was drawn towards the deployment of Docker containers [20]. As a result, a tremendous community has been established, providing a huge variety of Docker Images that are hosted on docker image repository (Docker Hub). All images have public access and can be pulled from the engine during deployment. Leveraging this feature, we constructed our images (docker-obc-airflow, netdata_nginx) that are essential for the Execution Environment and they pushed into DockerHub for public use. Additionally, utilizing the features that DockerHub gives, we have more capabilities when we use the Execution Environment in the cloud. To clarify, cloud providers like AWS, Microsoft Azure and Google already started including container technologies such as Amazon EC2, Google Container Engine and Azure Container Service and we can easily pull images through DockerHub.

3.3 Orchestration Tool

The orchestration platform organizes our container and constructs communication among them. The modeling of distributed applications for Docker-Compose including their lifecycle, dependencies, environment variables, and components are also defined using a YAML file. The YAML file is a human-readable data-serialization language. It is usually used for configuration files or applications such as Docker-Compose to define our dependencies. Below, we explain the docker-compose file per service with figures that integrated into our system.

Before we start the file explanation, we have to say that services have defined some values as environment variables that have been taken from a separate file (named: .env*)*. This file is a hidden file that can be found in the same directory with docker-compose.yml.

- **PostgreSQL**

```
postgres:
  image: postgres:9.6
  environment:
    - POSTGRES_USER=${POSTGRES_USER}
    - POSTGRES_PASSWORD=${POSTGRES_PASSWORD}
    - POSTGRES_DB=${POSTGRES_DB}
  ports:
    - "${EXECUTOR_DB_PORT}:5432"
  container_name: "local_executor_db_${EXECUTOR_INSTANCE}"
```

Table 2. PostgreSQL container into docker-compose.yml

In Table 2, we define the PostgreSQL, a powerful database that used to record information during the workflow execution. Is an image that was pulled from DockerHub. The other keys such as environment, ports and container name are definitions that are used for the container. In PostgreSQL we have to define username, password and database name, this information is implemented into the container as environment variables that are included in service definition. In the ports key we export the port that service is running, on the left side of definition is the port of your system, on the other side is the port that is allocated into the container. Finally, container_name key is to name the container that runs.

- **Monitoring System (Netdata) and Nginx**

```
netdata_monitor:
  image: manoskoutoulakis/netdata_nginx:latest
  environment:
    - ID=${NETDATA_ID}
  ports:
    - 19998:19998
  volumes:
    - /etc/passwd:/host/etc/passwd:ro
    - /etc/group:/host/etc/group:ro
    - /proc:/host/proc:ro
    - /sys:/host/sys:ro
    - /var/run/docker.sock:/var/run/docker.sock:ro
  container_name: "obc_resource monitoring"
```

Table 3. Netdata with nginx container into docker-compose.yml

In Table 3, we have another utilized image with two services. The first service is the Netdata for resource monitoring and the second service is the Nginx. The "volumes" keys are the host's directories that collect information for the system resources and are for read-only because this service can monitor the whole system not only the containers. For our purposes, the most significant volume is the final one that allows netdata to monitor Docker containers. The other keys that are defined are the same as the previous service.

- **Workflow Management System and Client**

```
airflowserver:
    image: manoskoutoulakis/docker-obc-airflow:1.10.9
    restart: always
    depends_on:
      - postgres
      - netdata_monitor
```

```
    environment:
      #Airflow configuration
      - AIRFLOW__CORE__SQL_ALCHEMY_CONN=
postgresql+psycopg2://airflow:airflow@postgres:${EXECUTOR_DB_PORT}/airflow
      - AIRFLOW__WEBSERVER__BASE_URL=http://localhost:8080/${OBC_USER_ID}
      - LOAD_EX=n
      - EXECUTOR=Local
      - FERNET_KEY=jsDPRErfv8Z_eVTnGfF8ywd19j4pyqE3NpdUBA_oRTo=
      #OBC Client environment variables
      - NETDATA_ID=${NETDATA_ID}
      - OBC_USER_ID=${OBC_USER_ID}
      - PUBLIC_IP=${PUBLIC_IP}
      - EXECUTOR_INSTANCE=${EXECUTOR_INSTANCE}
      - POSTGRES_USER=${POSTGRES_USER}
      - POSTGRES_PASSWORD=${POSTGRES_PASSWORD}
      - POSTGRES_DB=${POSTGRES_DB}
      - NETDATA_MONITORING_PORT=${NETDATA_MONITORING_PORT}
      - OBC_EXECUTOR_PORT=${OBC_EXECUTOR_PORT}
      - OBC_AIRFLOW_PORT=${OBC_AIRFLOW_PORT}
      - EXECUTOR_DB_PORT=${EXECUTOR_DB_PORT}
    volumes:
      - dagvolume:/usr/local/airflow/dags
      - logvolume:/usr/local/airflow/logs
      - reportvolume:/usr/local/airflow/REPORTS
      - /var/run/docker.sock:/var/run/docker.sock
    ports:
      - "${OBC_AIRFLOW_PORT}:8080"
      - "${OBC_EXECUTOR_PORT}:5000"
    command: webserver
    healthcheck:
      test: ["CMD-SHELL", "[ -f /usr/local/airflow/airflow-webserver.pid ]"]
      interval: 15s
      timeout: 15s
      retries: 3
    container_name: "executor_airflow_${EXECUTOR_INSTANCE}"
```

Table 4. Execution environment's API with the Airflow WMS container into docker-compose.yml.

The last service applied to the executable environment can be found in Table 4, it is more complicated than the other services. The reason that makes this container complex is that it must take into consideration all the information about other services, such as ports, database information, but also the Workflow Management System configuration. All this information is collecting using environment variables. There are two services integrated there, the Workflow Management System and the Client.

The client makes communication with the workflows repository (OpenBio) that generates the appropriate data and triggers the execution. The most significant part of this service is the volumes section. The volumes are persistent and save the data that have been created during the execution. These volumes have no relationship with the local system, only with the containers. When the execution has finished, the data can download into your system using the client's API. Also, we have to notice that we expose two ports to our local system, this occurred because of the usage of two services into the container. We have the *healthcheck* key that checks the container's health by running a command inside the container. This can detect crucial cases during the workflow's execution such as being stuck in an infinite loop or unable to handle the execution, even though the service process is still running.

The entire docker-compose.yml file and related files of the project are available at Github (https://github.com/manoskout/OpenBioC_Execution).

3.4 Docker Volumes

Docker Volumes are not controlled by the storage driver. Reads and writes to data volumes bypass the storage driver and operate at native host speeds. We can mount any number of data volumes into a container. Multiple containers can also share one or more data volumes. After mounting the container process writes to the specific directory of docker volume instead of writing directly on the host's filesystem. The advantages of this mounting are data is safe on Docker host by providing centrally located persistent storage and act as a central data storage facility that temporarily aggregates fragments of federated data for the need for analysis. Also, Docker volume preserves data regardless of the container lifecycle. These Volumes have specific directories in the Docker host and are created and managed by Docker itself. The volume is named according to the container's name or it is anonymous in case the container does not have a name.

```
manos@Ubuntu-1804-bionic-64-minimal:~/obc_executor_main$ docker volume ls
DRIVER              VOLUME NAME
local               eff156e7e73d899dc71561777b88e42888ace91bcb550d7d8aa21c06db9c35b0
local               obc_executor_main_dagvolume
local               obc_executor_main_logvolume
local               obc_executor_main_reportvolume
```

Figure 9. Execution Environment Docker Volumes

As we demonstrate in **Fig. 9**, the containers of the execution environment have generated 3 volumes that are used to preserve data, in case the system is interrupted unexpectedly. Firstly, volume named *dagvolume* keeps the workflows that are prepared for the execution process. Secondly, the *logvolume* saves the logs that the WMS records during the execution. Finally, the *reportvolume* keeps the results of workflow's execution. All of these are defined into a Docker Compose file.

4 Implementation

In this chapter, we provide a detailed description of the Execution Environment and OpenBio extension. We analyze obstacles encountered and try to put our design decisions into practice and the steps we took to overcome them. Lastly, we describe the methods and tools applied to assure the correctness of our code. The whole source code is available in GitHub (https://github.com/manoskout/OpenBioC_Execution).

4.1 Project Structure

The structure of the project is complex due to the usage of multiple tools and libraries. No one of the tools and libraries has to be pre-installed in our system except for Docker and Docker-Compose. The whole project's services are federated using docker-compose and installing all the dependencies into the containers. To put it briefly, the project contains HTML and CSS and JS for resource monitoring UI construction, Dockerfiles to build isolated environments for the services that are in use, Docker-compose file to organize and configure the containers, related configuration files to NGINX, Netdata and Airflow to parametrize our needs and Bash Scripts to make the installation files. The only file that is necessary for the integration of the Execution environment is the installation file.

Because of multiple tools and libraries, we encounter some problems during the project's development. We distribute all services into isolated images to avoid conflicts among the libraries, pushing them to DockerHub. Consequently, we had extended debugging information for every service separately thanks to Docker Compose. We finished our development by combining all required tools using docker-compose.

4.2 Containerizing Services

We already discussed the docker-compose file in the previous section with no reference for the Dockerfiles. Dockerfiles are the core of the implementation of our project. There were two Docker Images created for this thesis and only the most remarkable aspects of the ones created will be covered. A Dockerfile should be typically simplified as much as possible. For example, **Fig. 10** describes a very simple piece of Netdata-Nginx image using the Debian package manager APT. However, in the second container was Python Image which is Debian-based too but there are also different package managers such as Fedora which uses YUM etc. Basically, any tool can be containerized and the Docker will allow for these to run with no operating system restriction.

```
FROM debian:stretch-slim

WORKDIR /
RUN apt-get update && apt-get install -y --no-install-recommends apt-utils
RUN apt-get install -y \
    dnsutils \
    wget \
    uuid-dev \
    zlib1g-dev \
    gcc \
    make \
    autoconf \
    automake \
    pkg-config \
    libtool \
    libpcre3-dev \
    curl \
    libuv1-dev \
    nginx \
    fping
```

Figure 10. A base of our Netdata-Nginx Dockerfile for creating a Docker image with these services.

Installing and compiling from source is also possible in cases where the software is not available in the Linux core library. In **Fig. 11**, Dockerfile installs Netdata from source and Nginx from Linux core library. All the steps are identical to how the software would be installed on a local machine using bash, the only difference being the keywords are not included. At the end of the previous figure shows two important keywords called "COPY" and "ENTRYPOINT". The "COPY" as it is named copy the host's file or directory inside the container. The "ENTRYPOINT" executes the defined file whenever the container starts. Inside this bash script, we have defined some important configuration to run the webserver.

```
ARG NETDATA_UID=201
ARG NETDATA_GID=201
ENV DOCKER_GRP netdata
ENV DOCKER_USR netdata
RUN wget https://my-netdata.io/kickstart.sh
RUN chmod 777 kickstart.sh
RUN ./kickstart.sh --dont-wait
RUN \
    mkdir -p /var/log/netdata && \
    chown -R root:root \
        /etc/netdata \
        /usr/share/netdata \
        /usr/libexec/netdata && \
    chown -R netdata:root \
        /usr/lib/netdata \
        /var/cache/netdata \
        /var/lib/netdata \
        /var/log/netdata && \
    chmod 0755 /usr/libexec/netdata/plugins.d/*.plugin && \
    chmod 4755 \
        /usr/libexec/netdata/plugins.d/cgroup-network \
        /usr/libexec/netdata/plugins.d/apps.plugin && \
    find /var/lib/netdata /var/cache/netdata -type d -exec chmod 0770 {} \; && \
    find /var/lib/netdata /var/cache/netdata -type f -exec chmod 0660 {} \; && \
    ln -sf /dev/stdout /var/log/netdata/access.log && \
    ln -sf /dev/stdout /var/log/netdata/debug.log && \
    ln -sf /dev/stderr /var/log/netdata/error.log
RUN cp /etc/nginx/nginx.conf /etc/nginx/nginx.conf.backup
RUN cp /etc/netdata/netdata.conf /etc/netdata/netdata.conf.backup

COPY run.sh run.sh
RUN mv run.sh /usr/sbin/
RUN chmod 777 /usr/sbin/run.sh
ENTRYPOINT ["/usr/sbin/run.sh"]
```

Figure 11. The second piece of script of Netdata-Nginx Dockerfile.

In our case, third-party software is used, it can be difficult and not always efficient to know which essential dependencies are using Debian as a base image. In cases where third-party image

software is not used, a minimalistic base image can be used. For example, there are Linux distributions with a base image of size ~5Mb.

Like the previous Dockerfile, so in the second we follow the same structure but with some differences. First and foremost, we use Python as a base image. Subsequently, we set up some folders and files for the workflow management's data and the API. Also, another important configuration is that this container can handle our Docker platform, providing us more flexibility to leverage our workflow execution. Below, **Fig. 12** only shows the important part of the code that was implemented.

```
COPY script/entrypoint.sh /entrypoint.sh
RUN chown -R airflow: ${AIRFLOW_USER_HOME}
ARG DOCKER_UID
RUN \
    : "${DOCKER_UID:?Build argument DOCKER_UID needs to be set" \
    "and non-empty. Use 'make build' to set it automatically.}" \
    && usermod -u ${DOCKER_UID} airflow \
    && echo "Set airflow's uid to ${DOCKER_UID}"
RUN mkdir -m755 ${AIRFLOW_USER_HOME}/dags
RUN mkdir -m755 ${AIRFLOW_USER_HOME}/REPORTS
RUN mkdir -m755 ${AIRFLOW_USER_HOME}/REPORTS/WORK
RUN mkdir -m755 ${AIRFLOW_USER_HOME}/REPORTS/TOOL
RUN mkdir -m755 ${AIRFLOW_USER_HOME}/REPORTS/DATA

WORKDIR ${AIRFLOW_USER_HOME}
ENV FLASK_APP client.py
ENV FLASK_RUN_HOST 0.0.0.0

COPY /client/requirements.txt requirements.txt
RUN pip install -r requirements.txt
COPY /client/static/ ${AIRFLOW_USER_HOME}/static
COPY /client/templates/ ${AIRFLOW_USER_HOME}/templates
COPY /client/client.py ${AIRFLOW_USER_HOME}/
RUN mkdir -m755 ${AIRFLOW_USER_HOME}/logs

ENTRYPOINT ["/entrypoint.sh"]
CMD ["webserver"]
```

Figure 12. Airflow and Execution Environment's API Dockerfile.

4.3 Environment installation

The project was developed using an Ubuntu server. Thus, the first installation script was written in the BASH script and it is compatible with all debian-based distributions. The installation file defines some crucial information as environment variables and port allocation to facilitate the communication between the containers and integrate Docker and Docker Compose if necessary. The environment variables such as public IP, user unique IDs, and services' ports, database credentials are saved in a hidden file, provided if the installation succeeds. Then, the system downloads all the required data for the existence of the Execution Environment and following the installation instructions, the users need to copy the generated URL into the OpenBio platform to establish a connection between the platform and the execution environment. **Fig. 13** depicts the results when the installation succeeds.

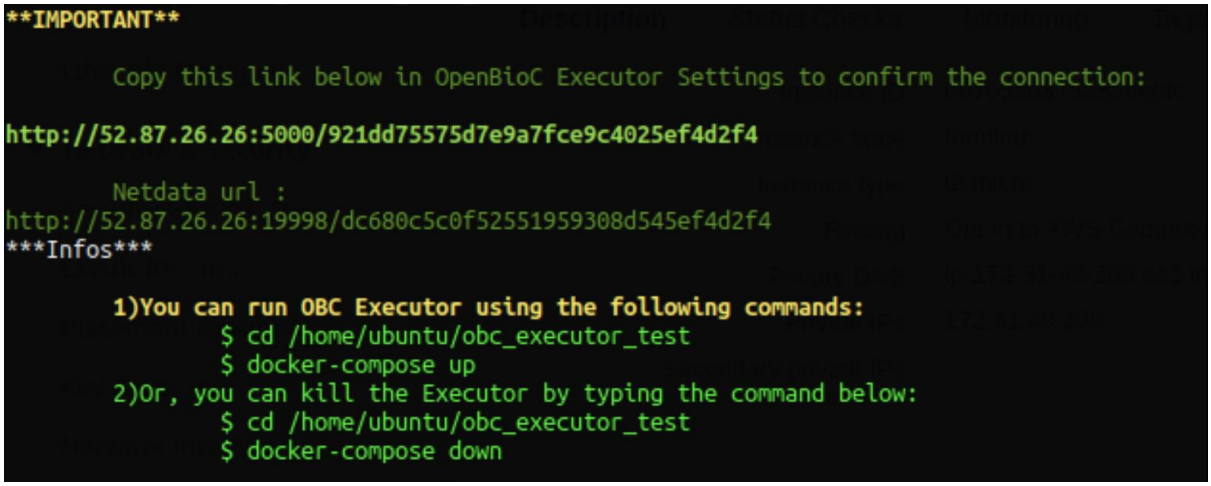

Figure 13. Successful installation output.

Besides the execution environment, we add another important tool named NetData for resource monitoring. Furthermore, another unique ID was generated in installation for security purposes for the Netdata recognizable only for you and our platform. Finally, the installation runs the constructed containers by running the docker-compose file to report if the system faces difficulties. As it shown in **Fig. 13**, we have a Netdata-URL which connects us on Netdata UI. Netdata provides a useful User Interface with crucial resources information not only for the execution environment, but also for the whole system.

4.4 Environment Features

The execution environment was developed as a web service to facilitate procedures such as creating, executing, and managing a scientific workflow. The developed service provides many features that are required. Nevertheless, more functionality can be easily integrated to the service. At the moment, the following capabilities were implemented:

- An interface according to the workflow management system to provision the scientific workflow during the execution.
- A plethora of compatible Workflow Management Systems which can be used to execute a workflow.
- A real-time resource monitoring dashboard that monitors the Workflow management systems that are used.
- Limited access to files, making input and output data invulnerable to attacks and used only for workflow purposes.
- A useful API that used to collaborate with the OpenBio platform. Apart from OpenBio platform, users can send, edit, delete, download results, and provision the execution.
- Automatically zip execution's results and logs for download.

As can be seen from the list above, the service consists of the three main parts: User, Workflow Management System, API to handle workflow. As we mentioned in the previous section, the whole service was integrated using Docker. Below we make an extended explanation of these features.

4.5 Workflow Management System Interface

In general, the Execution Environment works mainly as background service but facilitates its usage by providing to the users a flexible UI for workflow monitor. According to our needs, WMSs already provide user-friendliness UI and implement them into our execution environment. To clarify, all WMSs that tested our project had UI for workflow monitoring. Below we depict some examples of User Interfaces of WMSs. The Execution Environment uses the OpenBio platform to provide the process information and actions of workflow making the platform interoperable by facilitating the execution.

Fig. 14 shows us Luigi's user interface, using the web interface users can handle all the features that this WMS contains. Unfortunately, there are some restrictions such as the DAG of tasks cannot be viewed before execution. Thus, users wouldn't know what code is running in correlating tasks during deployment.

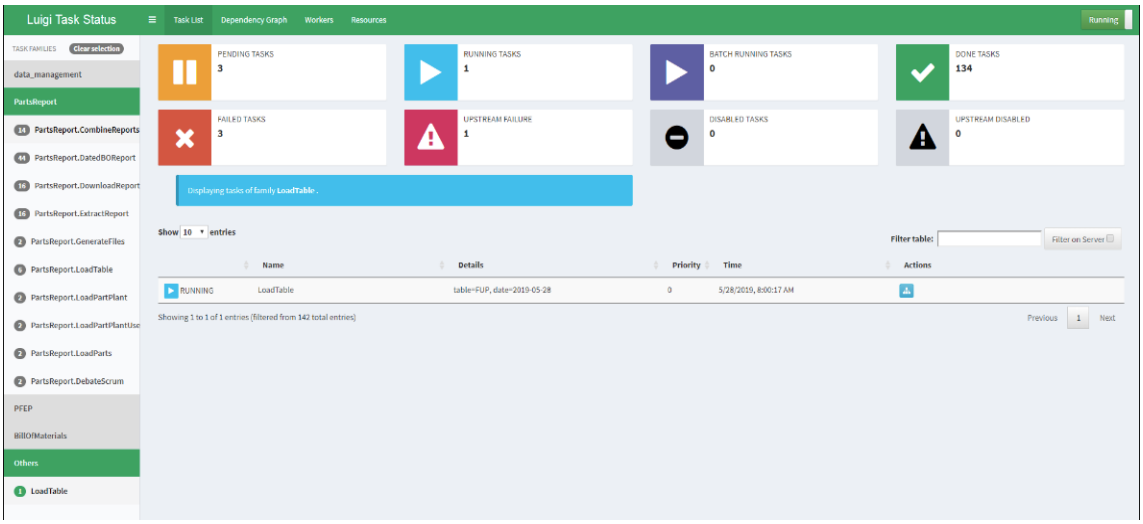

Figure 14. Luigi Workflow Management System User Interface

Next, **Fig. 15** depicts the Apache Airflow user interface. Contrary to Luigi, Airflow UI has a plethora of features such as Gantt Chart, Task Duration, Code View, Task instance content menu, etc. Contrary to these features, this WMS is not a preferred tool to execute bioinformatic workflows but is a great opportunity to implement it.

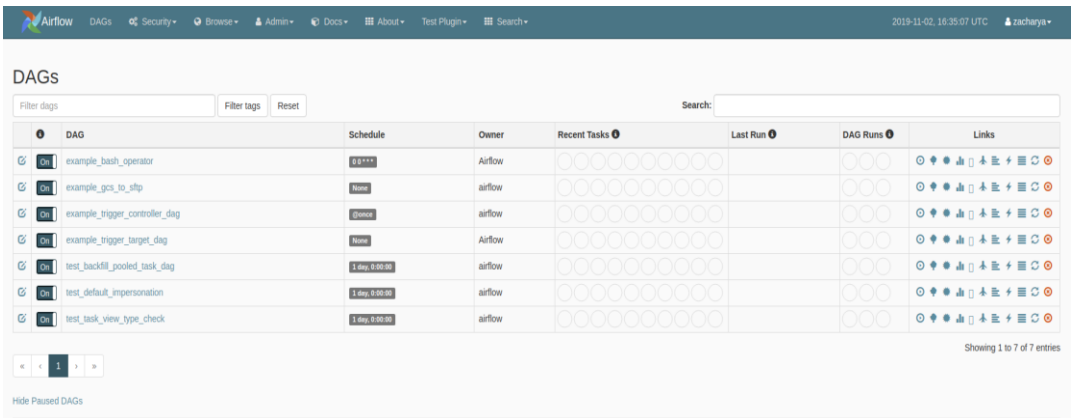

Figure 15. Apache Airflow Workflow Management System User Interface

4.6 Application Programming Interface (API)

An Application Programming Interface is a computing interface that manages the interaction between software mediators. It is a crucial factor, during the development making the software flexible and manageable. An API is a custom, and design based on industry to ensure interoperability. Usually, the term API is used to refer to the set of software entities that serve to implement the API of some encompassing component or system.

During project development, an API simplifies programming by isolating the underlying implementation, exposing only objects or actions that the developer needs. Below we show some examples of the features provided from our Execution Environment. Each of these tested using the CURL Linux command. To clarify, The API structure is not stable. We expect the endpoint definitions to change. Also, I would like to refer to the structure of these requests that were built according to the OpenBio platform needs and being collaborative with OpenBio User Interface.

4.6.1 ENDPOINTS

- **Trigger DAG from OpenBio Repository**

Executing this request, we receive the tool or workflow from the OpenBio platform as a dag file and automatically perform the workflow. This call requests a dag according to the data that gives. Hence, the table below shows us a POST request with data such as name, edit, type, callback, workflow_id.

```
curl --header "Content-Type: application/json" \
  --request POST \
  --data '{
        "name":"test",
        "edit":"1",
        "type":"workflow",
        "callback":"<Repository URL>",
        "workflow_id":"2"}' \
  http://<IP>:<Port>/<Unique ID>/run
```

Table 5. POST request to save and run a DAG

Name: We pass the name of the tool or the workflow that we would like to execute.

Edit: This is the specific version of the workflow from OpenBio. If the request is addressed to a tool, then the edit remains empty.

Type: There are two different types in OpenBio which are tools or workflows.

Workflow_id: This is a unique id auto-generated from the platform that we use or hardcoded from you.

At the end of this command, we have to specify the URL of our execution environment. During the installation of the execution environment in your system, you get a URL consisting of the IP of the execution environment, the Port that environment runs, and a unique ID which is known only from you.

- **Execution Status**

Obviously, everyone wants to know the status of their workflow during the process. Thus, we build a request providing extended information about the workflow. More specifically, this instruction inquires the database of what state is our workflow and returns a json object with the state and the current task that executes.

```
curl --header "Content-Type: application/json" \
  --request GET \
  http://<IP>:<Port>/<Unique ID>/check/id/<dag_id>
```

Table 6. GET request to get info about the status of the executed workflow.

id: This id is the unique id that was given when we triggered that dag. This is auto-created from the platform that we use.

- **Download the results**

In our environment, the data are separated into three different folders (Tool, Data, Workflow). At the end of workflow execution, all the results are collected and compressed to a tar file. Then, users can download the compressed file from the OpenBio platform or by requesting the specific id of the dag which they are interested in.

```
curl --header "Content-Type: application/json" \
  --request GET \
  http://<IP>:<Port>/<Unique ID>/download/<dag_id>
```

Table 7. GET request to download the results from a workflow execution.

- **Execution Logs**

Logging during the execution is an essential factor in debugging our workflow. During the execution, the WMS collects logs from the workflow's execution, compressing them into a zip file. In the OpenBio platform, users can get all the logs related to their workflows that are executed. The related request is written below.

```
curl --header "Content-Type: application/json" \
  --request GET \
  http://<IP>:<Port>/<Unique ID>/logs/<dag_id>
```

Table 8. GET request for the logs of the executed workflow.

- **Delete a workflow**

One more useful action in our environment is the deletion of a DAG. This request can be worked with DELETE or GET method. In other words, the user sends the delete request to delete the workflow. This action removes the workflow file, the database records from the workflow management system and all related files that had been created throughout the execution.

```
curl --header "Content-Type: application/json" \
  --request DELETE \
  http://<IP>:<Port>/<Unique ID>/workflow/delete/<dag_id>
```

Table 9. DELETE request to delete a workflow from the environment.

- **Executor Information**

Executor information such as failed, succeed, paused, running DAGs and workflow management engine information are provided in a get request using a data stream in real time. The update the statuses every 5 seconds. The usage of this endpoint is mainly for the monitoring system.

```
curl --header "Content-Type: application/json" \
  --request GET \
  http://<IP>:<Port>/<Unique ID>/executor_info
```

Table 10. Real-time workflows statuses streaming.

4.7 Execution and file access

Each workflow management system must assure that the assigned tasks must be executable. Hence, every single task depends on compiled binaries and libraries at the expected position into the file system to successfully proceed for execution. The most prominent ways of ensuring this are “virtualization” and “installation”. The first is an innovative way that facilitates the execution in an isolated container-based environment. Contrary to container-based virtualization, Virtual Machine is considered as a performance-harming method, requiring time-consuming programming to configure the environments. The second is the most usual way that requires the installation of a proper runtime environment in the operating system. By combining the container-based virtualization and the installation methods and leveraging Docker Volumes we built our environment. To facilitate these execution requirements, we constructed a simplistic file system structure into our Execution Environment container connecting into a specific persistent volume communicating with the OpenBio platform.

4.7.1 File System Structure and Docker Volumes

The file system structure is constructed according to the workflow type. A workflow is imported as a DAG in our environment. It could be a simple tool, a data collection, or a group of multiple tasks that contains tools, libraries, and so on. When the DAG is imported, we analyze the file and the execution starts automatically. Subsequently, the workflow management system informs us by providing extended logging records. Thus, the Execution Environment has three volumes each of them for a different purpose. All the above considered of the construction of these volumes as I/O (Input and Output) Manager. The events of the I/O Manager are handled from the API and OpenBio platform.

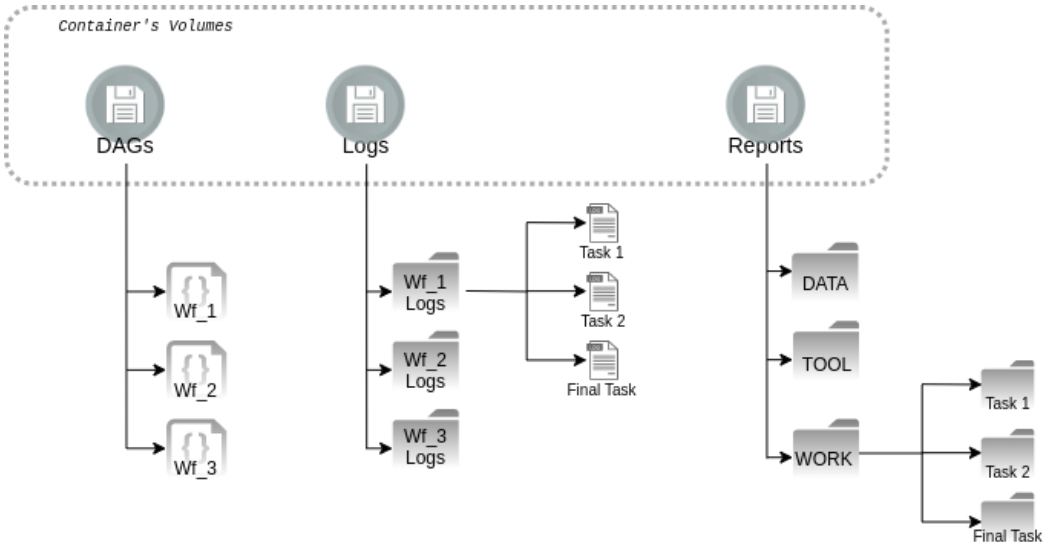

Figure 16. Container's Volume structure as a I/O Manager.

The first volume named DAGs saves the imported DAGs from the OpenBio platform to prepare them for the execution. The second volume is to persist the logs during the execution. Each of the log folders is a related workflow and saves the execution's logs per task. Finally, the volume named Reports provides the outputs of the executed workflow as well as the related tools or data that needed it. All the files used are grouped by the unique workflow_id that has been given from the OpenBio platform.

For the current work, it is sufficient to point out that we have not done extensive research on security issues. Although, volumes are inaccessible from the host, preventing unexpected attacks. The only way to edit or track the files can be achieved only from the OpenBio Platform.

4.8 Resource Monitoring Integration

Nginx server is an open-source, high-performance HTTP server and a reverse proxy tool. In general, the use of Nginx has centralized at web servers and it is commonly used as a load balancer managing incoming traffic. Nginx offers low resource consumption, simple configuration, and stability. Netdata is implemented in the environment for real-time metrics that provides to users (described in detail in Section 2.7). Nevertheless, the Netdata shares crucial information for the host over the Internet, making the system vulnerable to attacks from the internet. Thus, by using the reverse proxy we prevent unexpected attacks from the global network.

The reverse proxy provides an additional level of abstraction and control to secure the flow of network traffic between clients and servers. Also, reverse proxy control access to a server on private networks and it can perform cache or decrypt data. As I mentioned above, the system has been built with containerization technology and the Netdata with Nginx works perfectly. For additional security in this Docker image, Nginx has a unique ID auto-generated from the system throughout the installation that is set into the URL, providing the appropriate resource metrics during the workflow execution. The resource monitoring URL is reserved when the installation finishes or from the OpenBio platform. **Fig. 17** shows a basic diagram of how the NGINX works in our project.

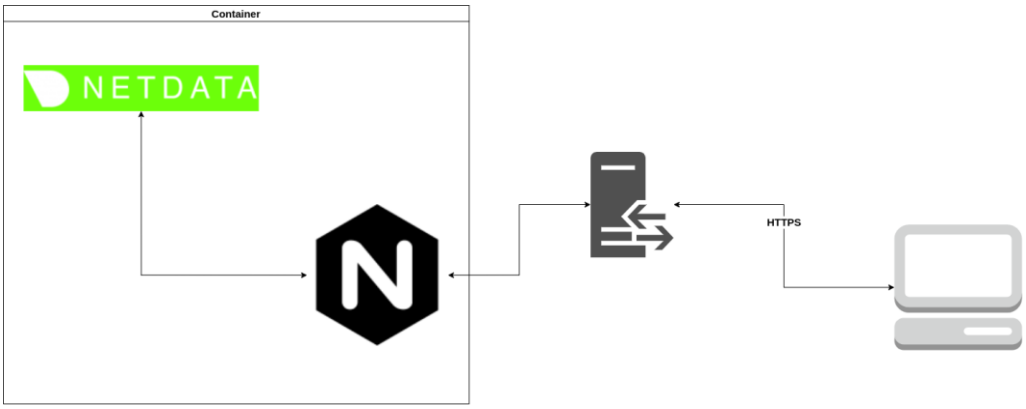

Figure 17. Netdata with NGINX.

5 Integration & Experimental Results

In this chapter, we describe an approach that we have used to integrate the Execution Environment with the Airflow [15] workflow manager. In addition, we introduce an example in bioinformatics to determine how widely-used large-scale data management infrastructure systems are in bioinformatics.

5.1 System Specifications

The system specifications used for the Execution Environment deployment are in **Table 11**. These specifications were sufficient to reproduce tests to monitor the robustness of our Environment. The only restriction we encountered was the CPU. As a result, the executions took place by using 2 and 4 cores.

CPU	RAM	DISK	Operating System
Intel Core i7-3770 CPU 3.40GHz, 4 Cores, 8 threads	32Gb	2.7 Tb	Ubuntu Server 18.04

Table 11. Host's specifications

In general, high-performance data analysis in bioinformatics demands faster CPUs as well as more RAM to run concurrently more than one complex workflow. Another crucial factor is the Hard Drive, during the execution, the workflow downloads plenty of datasets.

5.2 Execution Environment Integration

As I mentioned before, the system was tested only in Ubuntu Server. Thus, the installation of the Execution Environment can only be established in Debian distributions for the moment. The executable file written in Bash commands is readable and ready to install any tool that the Execution Environment needs. The installation is divided in three layers that each of them contains a sequence of bash commands. Below, we explain with simple instructions step-by-step how to deploy the Environment and connect it with the OpenBio Platform. Also, we show several pieces of each step that we have to consider such as useful ids or functions that used to create multiple instances in our machine.

5.2.1 Local Installation Steps

- Download install.sh

  The install.sh and whole project are published on GitHub. We download the installation file. In our case we use the "wget" library to pull the file from the repository. When the file will have downloaded, we execute the executable file.

```
$ wget
https://raw.githubusercontent.com/manoskout/OpenBioC_Execution/master/obc_scripts
/install.sh
$ bash install.sh
```

Table 12. Installation bash commands.

- Docker Installation (First Step)

  When the installation starts, the first operation that is done is to install Docker. To prevent override problems such as multiple Docker platforms the system checks if the Docker is preinstalled. If the Docker is installed, bypass this step and continue on the next step. Otherwise, the installation continues the docker installation.

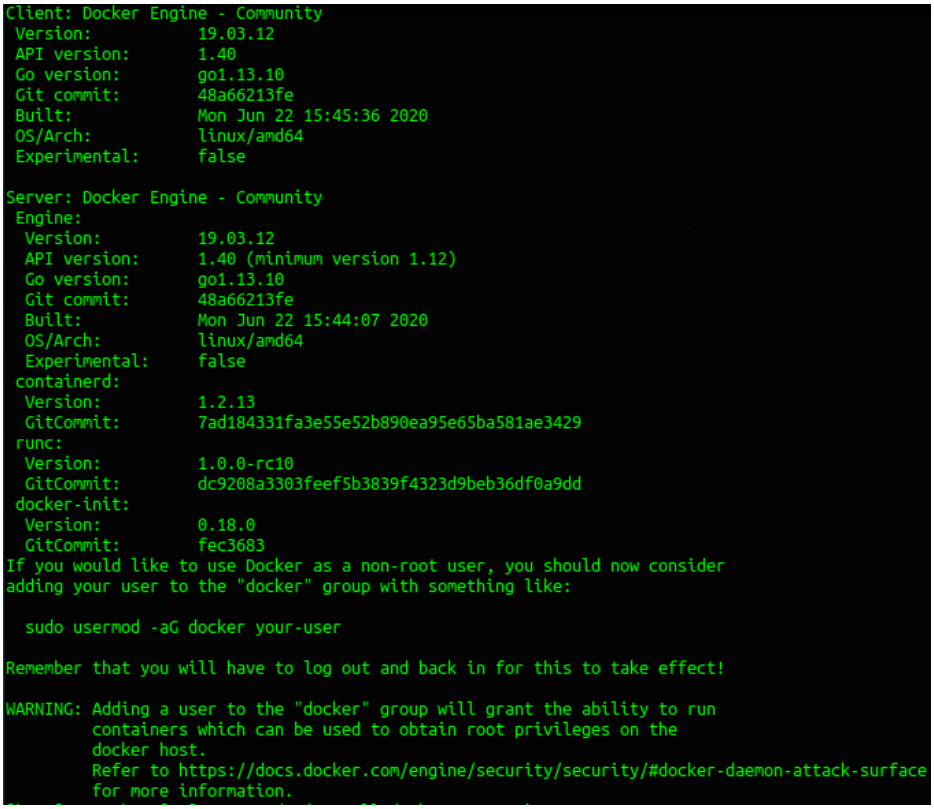

Figure 18. Installation step 1. Docker Installation.

- Docker-Compose (Second step)

  The second step is akin to the previous step. More specifically, the system checks if the Docker Compose is installed in our system. Correspondingly, if the Docker Compose is preinstalled, bypass the installation and executes the final step or else the installation of Docker Compose starts.

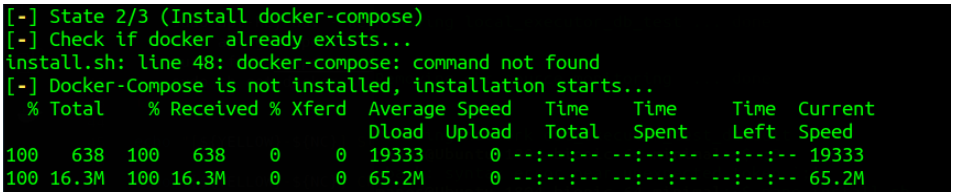

Figure 19. Installation step 3. Docker-compose Installation.

- Setting up environment variables and OpenBio Executor installation (Third step)

The final step is the most essential in our installation process. In this step, the installation defines some environment variables that are necessary for communication between OpenBio. The system examines the ports that are needed for the services. If ports are in use, automatically indicates the next ports of the host from the built function inside of the installation file. Fig 20 depicts the installation during Step 3. The only input required is the Execution Environment's name. To put it briefly, this name must be unique because it can be more than one Environments integrated into the host.

```
[-] State 3/3 (Setting up variables and installing the OpenBio Executor)
[\033[1;32m-->\033[0m] Enter your executor name: test
[\033[1;33m-\033[0m] Executor name is set : 'test'
[-] Set installation path on your environment: /home/ubuntu/obc_executor_test
[-] Client ID for OpenBioC Server : b2a6602aa47c7bc2fb13d6535ef4e585
```

Figure 20. Installation step 3. Insert Execution Environment name.

As the name is set, the installation process creates unique IDs for the OpenBio server and Netdata service. These IDs are utilized as the only way to communicate our machine with the platform. Therefore, the IDs are unique and are only known by OpenBio and the users. Besides these, the installation collects and generates other environment variables such as Database credentials, host's public IP, and the ports that run services. Importing environment variables, we increase the interoperability by trading plenty of information between the services. In **Fig. 21** is the environment variables' hidden file that contains all the required environment variables into the containers.

```
EXECUTOR_INSTANCE=test
POSTGRES_USER=airflow
POSTGRES_PASSWORD=airflow
POSTGRES_DB=airflow
OBC_USER_ID=b2a6602aa47c7bc2fb13d6535ef4e585
PUBLIC_IP=54.209.174.180
OBC_EXECUTOR_PORT=5000
OBC_AIRFLOW_PORT=8080
NETDATA_MONITORING_PORT=19998
EXECUTOR_DB_PORT=5432
NETDATA_ID=0c6a5b5ed685d6f5fc794baf5ef4e585
```

Figure 21. Hidden file with environment variables

The installation continued by downloading the docker-compose.yml file that is responsible for developing the executable environment. Also, the installation process downloads another file that configures workflow management. As we mentioned above, the WMS used is airflow. Thus, the configuration file to set aside the required WMS. This process downloads and installs the images, making the configuration that docker-compose file has.

```
Creating network "obc_executor_main_default" with the default driver
Creating volume "obc_executor_main_dagvolume" with default driver
Creating volume "obc_executor_main_logvolume" with default driver
Creating volume "obc_executor_main_reportvolume" with default driver
Creating obc_source_monitoring  ... done
Creating local_executor_db_main ... done
Creating executor_airflow_main  ... done


 Successful installation


Close tests ....


Stopping executor_airflow_main  ... done
Stopping local_executor_db_main ... done
Stopping obc_source_monitoring  ... done
Removing executor_airflow_main  ... done
Removing local_executor_db_main ... done
Removing obc_source_monitoring  ... done
Removing network obc_executor_main_default
```

Figure 22. Building the services and checking for conflicts.

Finally, the last output has all the information that we need to implement into OpenBio and how to run the Environment. The only action that remains is to add the host into the platform. **Fig. 23** depicts the steps that we have to follow to connect the Environment with OpenBio.

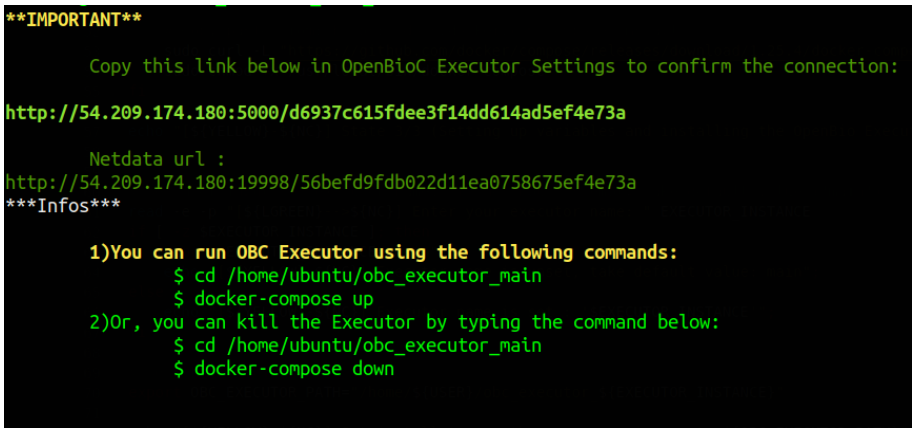

Figure 23. Last output of the installation with instructions.

5.2.2 OpenBio Connection

In this section, we represent the final guidelines for the communication between OpenBio and Execution Environment. The OpenBio platform has a simplified UI that makes the deployment easier. Below, we show an example of this deployment.

- **Connect to the Platform**

First and foremost, we must follow the previous instructions to build the environment into our host or cloud provider to perform the connection between the server and Execution environment. Then, we should open our browser and sign in to the OpenBio Platform (https://www.openbio.eu/platform/). If we don't have signup, we have to do that before we continue. Then, we navigate to the *User Profile setting -> Execution Environment* to add the Execution Environment's URL into the platform. **Fig. 24** depicts the inputs that needed to make the integration.

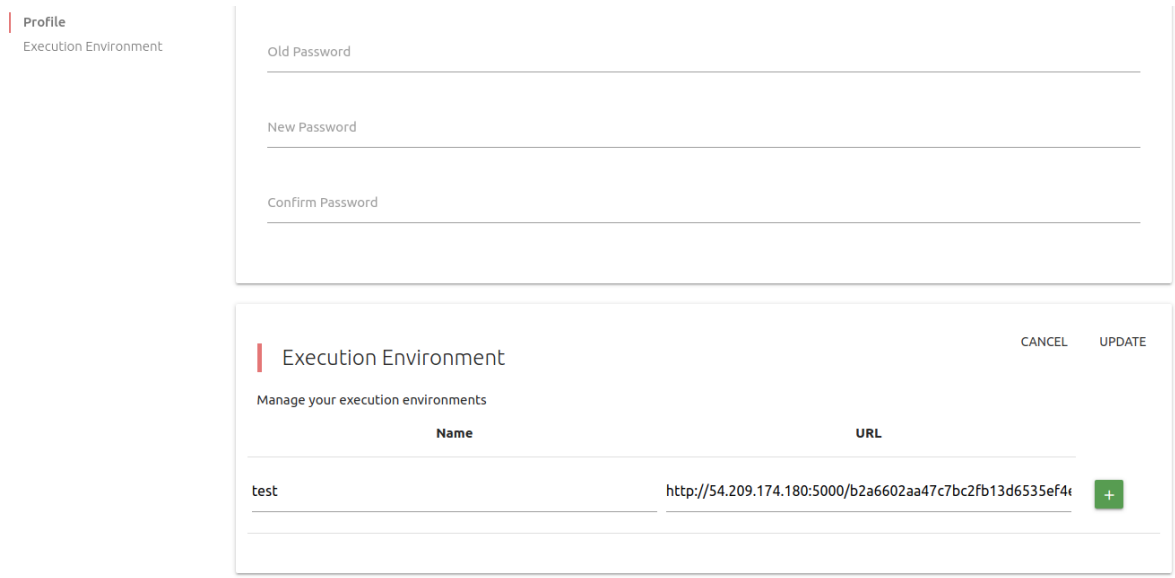

Figure 24. OpenBio Profile settings page.

At this point, we should insert some information. The one is the name of the environment, and the second is the URL containing the unique ID. As we mentioned before, the unique id has already created the installation complete. When we insert these two inputs, we click the plus button next to them and the environment is defined into the OpenBio platform.

5.3 Experiment

In this section, we present an experiment that was designed to estimate the efficiency of the proposed environment. It describes a workflow and the experimental environment used to perform the test. This evaluation methodology is designed to validate the overall proposed approach and its key components such as the workflow reproducibility, workflow provenance comparison and execution environment's interoperability. It then discusses the identified experiment, resources that consumed and expected output that will be discussed and analyzed in Subchapter 5.4. It also provides detailed information about the test environment and the workflow management system used to perform the experiment in order to validate the work carried out in this dissertation.

5.3.1 Execution Infrastructure

To carry out the experiments and workflow execution on the Execution Environment, a host-based infrastructure was used in this research study. Airflow has been used as a workflow management system to submit and monitor workflow execution. The workflow execution took place on a local computer, using the Airflow as workflow management service and Netdata as a resource monitoring service. This infrastructure uses Docker to offer SaaS services. To support data over a Docker-based storage service, docker has created virtual volumes. Since OpenBio supports RESTful interfaces, this service can also be called from RESTful clients. Using this API, the OpenBio can interact with Execution Environment's compute and storage services.

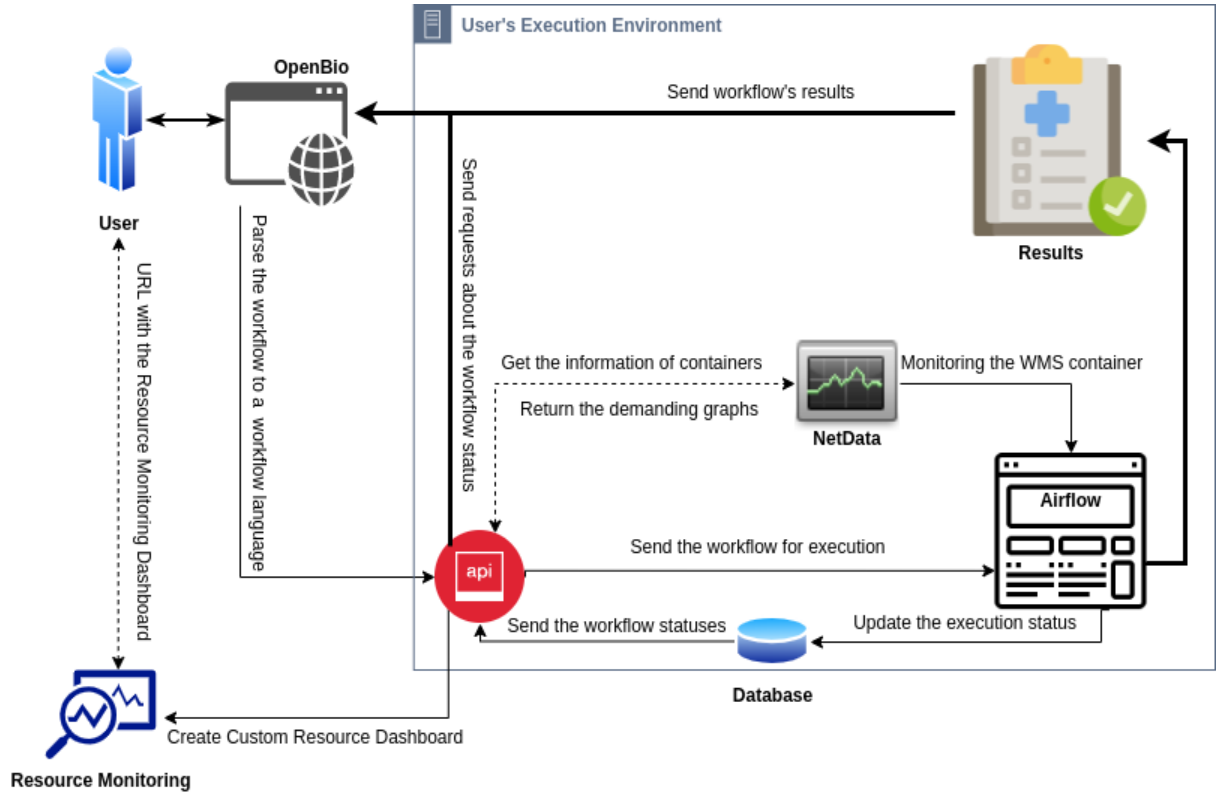

Figure 25. Execution Environment. Flow of operations.

Fig. 25, shows the steps performed during execution. The steps to perform the workflow that make the Execution Environment are:

- Firstly, it is obvious that the users have followed the instruction of how to install the Execution environment into their system and add it to the OpenBio platform.
- If the environment is in action, the Resource Monitoring Dashboard is running automatically. Users can have access to the custom dashboard that monitors the workflow management service. Alternatively, they can use the URL with the unique id from the installation output (see Figure 23).
- Users select the workflow that they would like to execute and the Execution Environment from the OpenBio platform.
- Then, OpenBio parses the workflow into a specific workflow language according to the workflow management service that the users have integrated into their execution environment.
- The execution environment's server gets the request containing the workflow with a unique ID provided from the platform.
- The server pushes the workflow to the Workflow Management Service for execution and sends a response to OpenBio.
- When the execution starts, the user can handle and get the status of the execution from the platform. The actions that are provided from the platform are the resource monitor, execution monitor, execution status, and delete or pause the execution. Concurrently, the database starts to update the status of the workflow in real-time.
- Once the execution is complete; the results and execution logs are saved into persistent container's volumes and they can be downloaded from the platform from the new buttons that are generated when the status is "success" or "failed".

5.3.2 Workflow explanation

The main purpose of this workflow is to build a scatter plot according to the HapMap dataset using Principal Component Analysis (PCA). As shown in **Fig. 26**, the main workflow is the *hapmap3_pca/1* which calls another sub-workflow named pca_plink_and_plot/1. Subsequently, the pca_plink_and_plot/1 call two more workflows the pca_plink/1 and the 2d_scatter_of_plink_pca/1. The workflow finishes by creating a scatter plot as a report. Below we make an extended reference about the workflow's tools that were used to implement this workflow in our execution environment.

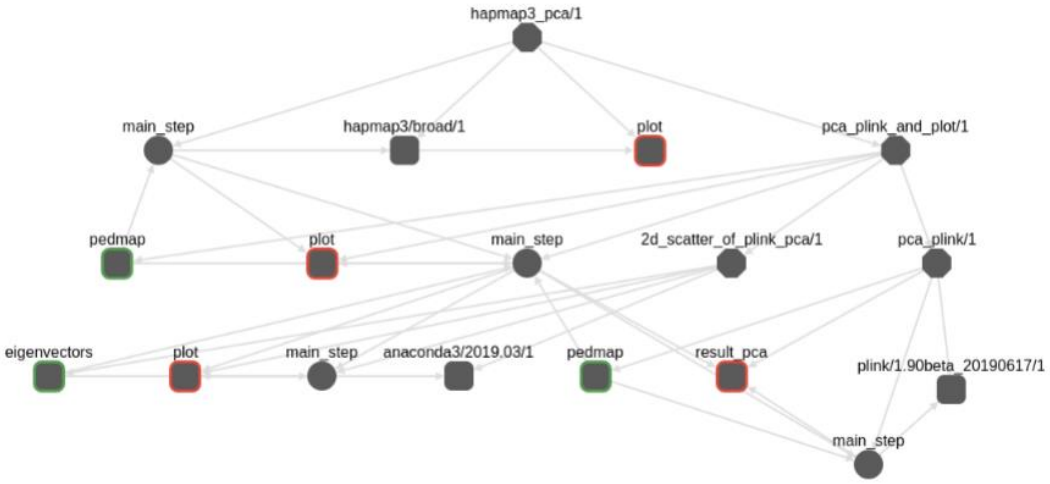

Figure 26. Hapmap dataset to PCA scatter plot. Workflow Graph

- **Hapmap3**

The HapMap (Haplotype Map) [30] is a dataset of common genetic variants called single nucleotide polymorphisms (SNPs). Every one SNP depicts a variance in a single DNA building block called a nucleotide. These variations eventuate normally in every part of a person's DNA. When several SNPs grouped together on a chromosome, they are inherited as a haplotype. The HapMap traces out haplotypes, including their locations in the genome and how common they are in different populations all over the world. The tool named *hapmap3/broad/1* that downloads the dataset is called from the main workflow (*hapmap3_pca/1*).

- **Plink**

Plink [31] is an open-source whole genome association analysis toolset, designed to perform a range of basic, large-scale analyses in a computationally efficient manner. It focuses on analysis of genotype/phenotype data. This tool is performed from the *pca_plink/1* workflow. To put it briefly, plink executes the hapmap dataset using Principal Component Analysis and returns the eigenvectors to prepare the construction of the plot.

- **Anaconda**

Anaconda [32] is an open-source data science toolkit. It provides a wide range of libraries. Under our circumstances, we used the NumPy library to construct a scatter plot of the PCA analysis. More specifically, this step gets the eigenvectors from the previous step and creates the scatter plot. Finally, the workflow ends by auto generating a report containing the scatter plot as an output of the workflow.

The workflow was written in bash using the OpenBio platform. Then, the platform parsed this workflow and converted it to airflow DAG to make the execution into the Execution Environment. We provide this workflow in OpenBio Repository (https://www.openbio.eu/platform/w/hapmap3_pca/1).

5.3.3 Workflow Execution

As we mentioned above, we assume that we have already created an account and we have added the execution environment into OpenBio (we provide extended information in previous sections). Also, we have to create our own workflow or use an existing workflow from other users. The first phase of the execution is depicted in Fig 26, as shown, we have chosen the workflow and by hitting the Run button it opens a dropdown menu. This dropdown menu contains the execution environments we have added into the platform. We choose the *test_thesis* (the specs of this environment are depicted in Table 11).

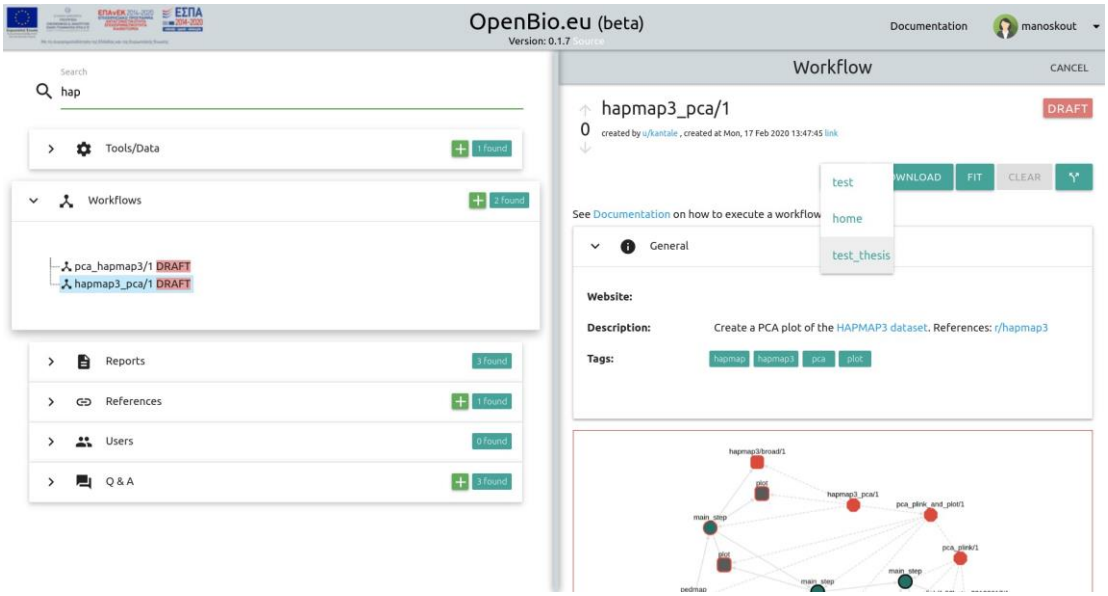

Figure 27. Openbio platform. Execution Environment selection.

When the execution begins, the Platform informs us that the execution was sent to our execution environment providing notification right up square of the platform. In case of error, the platform provides the error to us with a possible solution. Furthermore, a Report id is generated automatically and a new report is created on the left side of the platform in the Reports catalog. Fig 27 illustrates a successful submission for execution.

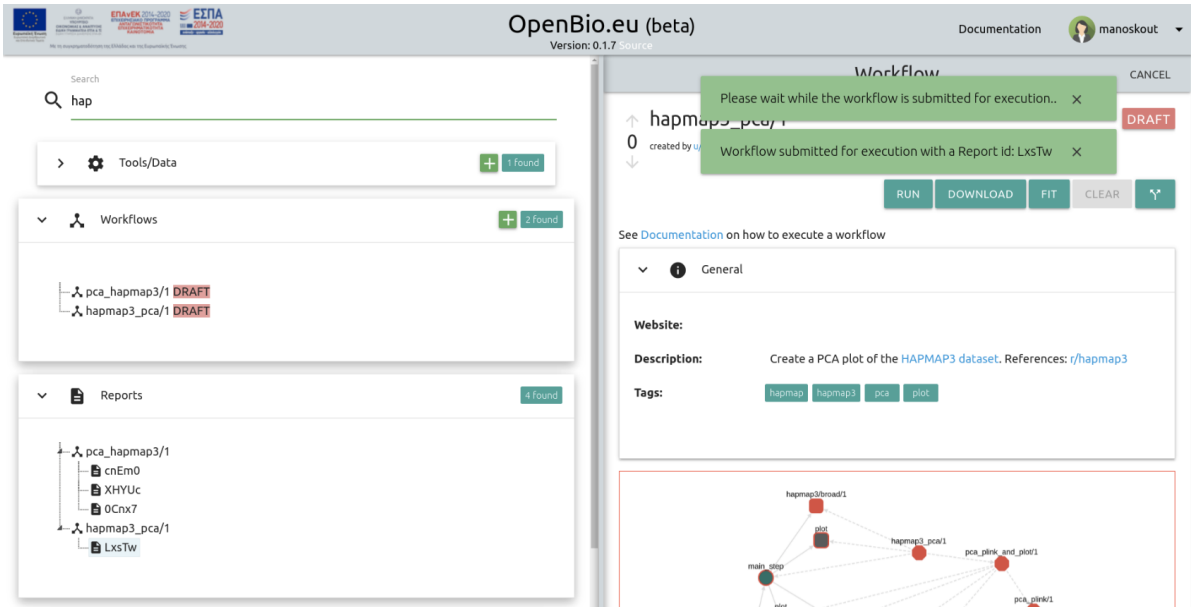

Figure 28. OpenBio. Successful submission for execution.

When the execution begins, we obtain the id of the workflow, by selecting this Report Id that the workflow has on the left side of the platform (more information in Fig 27), we can have access to the workflow execution. By clicking the unique id, the right side of the platform changes, and the workflow controller takes place. In this phase, we can pause, delete, or take the status of the workflow. These operations facilitate the execution because we do not implicate the WMS or other intermediate configurations.

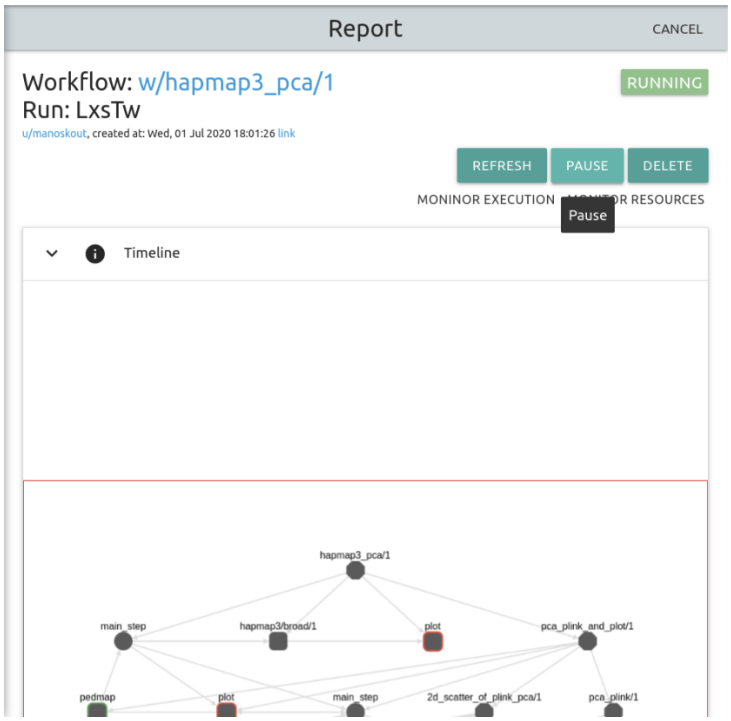

Figure 29. Workflow report and control panel.

Although, the OpenBio, provides us two more buttons as shown in **Fig. 30**. The first button is the Monitor Execution, which redirects us in the execution environment's WMS if we would like to make additional configurations. The second button is the Monitor Resources, which

redirects us in a custom dashboard that collects information from the Netdata Service which is integrated into the execution environment and general information for the workflows that the system has.

Figure 30. Workflow Successful execution.

By clicking the refresh button, the platform communicates with the execution environment to collect information about the execution. The final status that a workflow implements is a SUCCESS or FAIL. In Fig 29, the workflow executed successfully. As a result, two more buttons were shown when the execution finished. Aforementioned, the workflow's output is integrated into the HTML that is available if we click the Report button. We also provide logs that facilitate the debugging of the workflow mainly. We can download the logs by clicking the Logs button. Nevertheless, the Delete button remains in the foreground and the users could delete the report whenever they would from the OpenBio and the execution environment permanently.

Below, we provide some figures containing the compressed file, the results and the report of the workflow.

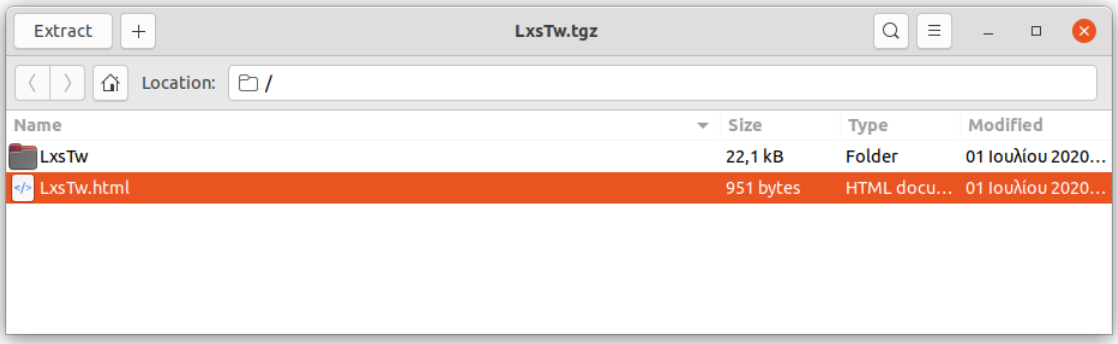

Figure 31. The compressed file of the workflow report. When we download the compressed folder there is the report and the outputs inside the file.

Figure 32. Workflow's Report. This HTML file contains the inputs and the outputs of the workflow. Also, we can have access to the outputs by clicking them.

As we mentioned before, the Report is constructed from the workflow as an integrated step. Report provides useful information such as the inputs and the outputs of the workflow. Also, we can have

access to the report file from the HTML file, because it uses tags that redirect us to the file that we choose. In **Fig. 33**, the output of the workflow is a scatter plot constructed from the workflow.

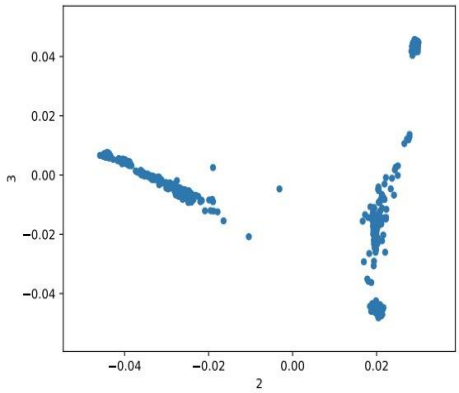

Figure 33. The scatter plot (Workflow's output).

5.4 Results

The HapMap PCA analysis takes approximately 9 minutes to run on a server with 8 cores clocked at 3.40GHz and 32Gb of memory. The PCA is computational unit intensive. Also, the workflow tested with parameterized resources which are represented below. In **Fig. 33** we visualized the performance of the previous workflow with different resource allocations. As expected, an increase in the number of CPUs and amount of memory to the WMS decreases the execution time. The time it takes for the workflow to start the first time is approximately 30 seconds which is not considered in this comparison. After the first deployment the system is much faster (~ 15 seconds) because of the cache of the WMS. Furthermore, if the tool that contains the workflow is already installed from another workflow the execution times decreased drastically.

The time it takes for the workflow to finish reaches over 8 minutes at best but is then saturated. A big increase in performance is seen when comparing the machine with 4 core vs the eight cores. It is almost twofold decrease in time, now given a larger dataset might take days to run, a two-fold decrease in execution time is good.

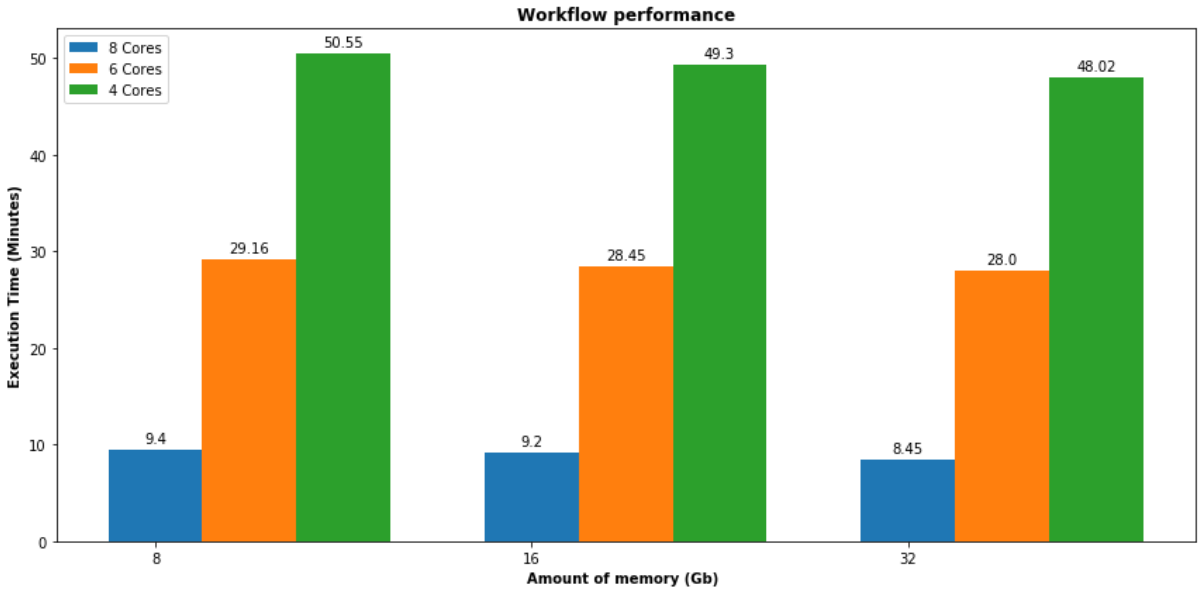

Figure 34. Workflow Performance of the Execution Environment measured in minutes with different machine setups. Each triplet of bars has a different amount of memory. The general trend being that the execution time decreases as the resources increases which is expected.

The results of the workflow performance were tested using Airflow as WMS. We expect that the results will vary when we use different WMS. The main purpose of these tests is to check the consistency of the workflow by performing heavy computational tasks. Because of the lack of process units, we decided not to testmore than eight cores. Although, for time sufficiency the more cores we have, the less execution time is.

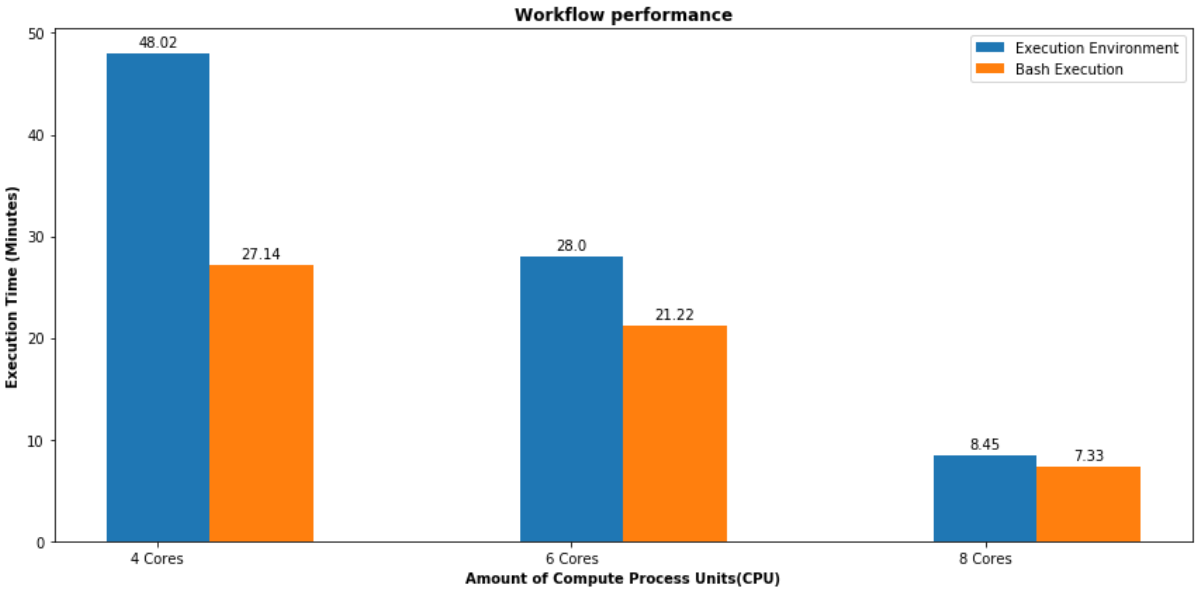

Figure 35. Workflow Performance. Comparison between Bash Execution and Execution Environment.

Additional tests, such as comparing the execution environment using a workflow management system and the execution environment using bash script have radical differences especially in terms of time. As it shown in Fig 34, the bash execution is beneficial according to the execution time but difficult for debugging. The bash execution lacks a proper workflow monitoring and logging throughout the execution. More specifically, using a WMS we have extensive information about the status and the progress of the execution. Nevertheless, we could leverage

the bash execution using container-based workflows. Containerized workflows are one of our priorities for integration into the execution environment.

The results are not only relative to the system's performance, but of the users' facilitation to execute a workflow in different environments. To put it briefly, if we compare the difference between execution time and the cores that are set, the results are varied. The more cores there are, the faster the execution is. Consequently, the difference in execution time, if we have more than eight-core, is a minor according to the benefits that we earn using the execution environment with the workflow management system of our choice. The main purpose of the Execution Environment is to run complex scientific workflows as well as extended workflow monitoring. Also, without a doubt, Bash execution can not co-work with platforms such as OpenBio and it does not have the interoperability that the Execution Environment has.

6. Conclusion

In this final chapter, we present a brief synopsis of our work assessing some principal points of the design process. Following that, we conclude by mentioning a few possible extensions and improvements that could be developed in the future.

6.1. Concluding remarks

The primary goal of this thesis was to implement and design a system covering our rationale to support multiple workflow management systems into one execution environment. Despite that, there was one more underlying goal: to effectively cooperate with developers and OpenBio users. We may now conclude that both objectives were met.

Regarding the main goal, the Execution Environment extension into the OpenBio platform, we achieved a breaking change: OpenBio platform can now execute multiple types of workflows using cloud, host or clusters. As proof of this concept for our implemented rationale, there are several reports into OpenBio repository which were created using the Execution Environment.

As far as the second goal is concerned, the collaboration was a first-time experience including constructive stress and the integral support from my supervisors by providing useful advice about my thesis purposes.

On this basis, we conclude that the Workflow Management Systems (WMS) allows life science communities to collaborate to make scalable and portable scientific research. The combination of multiple WMSs into one environment which communicates with a Bioinformatician Repository, without a doubt, brings a lot of benefits. This thesis proposed many resolutions to problems such as workflow adaptability and flexibility over the life science communities. The proposed environment of the workflow execution works through the interaction with users by logging in to the OpenBio platform.

All in all, this project aims to facilitate scientific research, providing a scalable and interoperable execution environment for sharing and publishing scientific research. The execution environment provides extended information about the workflow execution and it could work perfectly at any platform or repository such as OpenBio, because of the operable API it provides.

6.2. Lesson Learn

Nowadays, computational science demands a high-performance infrastructure that can be able to run complex workflows [1,2]. With the term of complex workflows, we mean workflows that integrate multiple methods such as programs and services from different organizations or algorithms, and high-throughput data and other components that are orchestrated as steps in a workflow [31].

Workflow Management Systems (WMS) lay the foundation for data and biomedical research. The main benefits of workflow execution by using a WMS are the effectiveness, reproducibility of procedures and traceability [32-35]. As we mentioned in previous sections, a

tremendous number of WMS are available in public, collaborating with the scientific research. Although, WMSs have limitations and they reduce their impact in biomedical research since each WMS has its own workflow language, a frustration for the users and demand for additional programming knowledge.

In this thesis, we provide a comprehensive presentation of different issues that directly affect interoperability among the execution of scientific workflows, except for the performance results that make the difference. The insertion of multiple workflow execution engines into one execution environment could diminish the "lock-in" syndrome [36], making the workflows reusable, accessible for users with no additional programming knowledge rather than BASH commands only. Additionally, multiple organizations and workflow system vendors have proposed a user-friendly workflow language called Common Workflow Language (CWL) [37] aiming to promote portability of workflow specifications. This workflow language has already integrated into our environment and it can work with many workflow management systems [38].

6.3. Current limitations

By the time, the execution environment had integrated Airflow as a WMS, t a widely used pipeline engine. The support of other workflow execution engines such as Snakemake [19], Nextflow [9], Luigi [8], and CWL [37] based workflow execution engines would increase the functionality of the execution environment. The Execution environment's structure has developed and prepared for further WMSs addition and this is the main reason for the implementation of this thesis.

According to execution time results, the bash execution is better than the WMS execution. In order to balance the execution time, we have to integrate lightweight WMSs or edit the existing execution scheduler. This is a minor limitation according to the benefits that a workflow execution engine provides like debugging from the execution logs and the extended execution monitoring. Nevertheless, we did not test with different WMSs to perform a complete comparison with other workflow engines.

In general, scientific workflows do not have loop conditions. Many proposals have presented the theoretical background of this abstraction [43,44], with no implementation in practice. This is a severe limitation and reproduces problems such as time and resource-consuming [45]. This is an exquisite topic for improvements and it is one of our future work, the implementation of conditions in scientific workflows.

6.4. Future work

Although we have implemented a couple of enhancements through the OpenBio platform's User Interface upgrading the overall User Experience we should make radical improvements into the Engine's core. Also, there are crucial additions to the Execution Environment's API and that makes the workflow execution flexible.

In addition, we have to integrate more workflow management engines such as Galaxy [7], Nextflow [9], Taverna [12] which are specifically developed for the field of bioinformatics. This could robust the portability of scientific research by providing to the users a plethora of workflow execution engines from only one service. The Execution Environment's design is adaptable and it can facilitate the integration for additional workflow management engines.

Another future work is Kubernetes integration. Kubernetes [39] is a platform for container and services management that facilitates the configuration and automation. It has a rapidly expanding ecosystem and supports widely available tools. Kubernetes can solve several problems by providing a framework to run distributed systems resiliently. More specifically, it takes care of scaling requirements, failover, deployment patterns, and more. Also, the Kubernetes platform provides service discovery and load balancing, self-healing to improve the Execution environment rejuvenation [40] (restarts the containers that fail), and storage orchestration.

Last but not least, an intermediate layer of this project could be a job manager. In general, a job manager is a resource management system which controls program execution of jobs in the background on supercomputers, clusters, and grids. The resource management system can manage jobs that users submit to various queues on a computer system. Under our circumstances it could be useful if we can parametrize every step of a workflow under our demands. There are many job managers such as Globus [41] and Torque [42].

Finally, as shown in the results section, the workflow can be faster by executing processes using bash rather than using a workflow management system. By leveraging the simple bash and running the workflow into a container we could earn a lot of benefits about the execution time efficiency. On the other hand, it cannot be denied that using a workflow management system we lose severe advantages such as workflow execution status, extended workflow logging, and portability.

References

[1] Alyass A., Turcotte M. and Meyre D. (2015) *From big data analysis to personalized medicine for all: challenges and opportunities.* BMC Med. Genomics, 8,33.

[2] Muir P., Li S., Lou S., Wang D., Spakowicz D.J., Salichos L., Zhang J., Weinstock G.M., Isaacs F., Rozowsky J., et al. (2016) *The real cost of sequencing: scaling computation to keep pace with data generation.* Genome Biol., 17, 53.

[3] (2015) *Top 5 benefits of server* (Accessed on 14/4/2020). [Online]. Available: https://content.dsp.co.uk/top-5-benefits-of-server-consolidation

[4] Docker. (2020) *Why docker?* (Accessed on 15/4/2020) [Online]. Available: https://www.docker.com/why-docker

[5] Docker. (2020) *Overview of Docker Compose* (Accessed on 15/4/2020). [Online]. Available: https://docs.docker.com/compose/

[6] (2014) *Docker vs VMs*. (Accessed on 14/4/2020). [Online]. Available: https://devops.com/docker-vs-vms/

[7] Afgan E., Baker D., Batut B., Beek M., Bouvier D., Cech M., Chilton J., Clements D., Coraor D., et al. (2018) *The Galaxy platform for accessible, reproducible and collaborative biomedical analyses.* 2018 update., Nucleic Acids Research, Vol. 46, Web Server issue W537–W544.

[8] Luigi. (2020) *Philosophy* (Accessed on 18/4/2020) [Online]. Available: https://github.com/spotify/luigi

[9] Di Tommaso P., Chatzou M., Floden E., Barja P., Palumbo E., et al. (2017) *NextFlow enables reproducible computational workflows.* Nat Biotechnol 35: 316-319.

[10] Krieger T. M., Torreno O., Trelles O., Kranzlmuller D. (2015) *Building an open-source cloud environment with auto-scaling resources for executing Bioinformatics and biomedical workflows.* 2. Related Work (Workflow management systems) pp. 5

[11] Badia R., Ayguade E., Labarta J. (2017) *Workflows for science: A challenge when facing the convergence of HPC and big data.* Supercomput. Front. Innov.: Int. J., 4(1):2747

[12] Wolstencroft K., Haincs R., Fellows D., Williams A., Withers D., Owen S., Soilanand-Reyes S., Dunlop I., Nenadie A., Fisher P., et al., (2013) *The Taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud.* Nucleic acids research.

[13] Google Cloud (2020) *What are containers?* (Accessed on 24/4/2020). [Online], Available: https://cloud.google.com/containers

[14] OpenSource (2020) *What is docker?* (Accessed on 24/4/2020). [Online], Available: https://opensource.com/resources/what-docker

[15] Apache Airflow Documentation (2020) (Accessed on 25/4/2020). [Online], Available: https://airflow.apache.org/docs/stable/

[16] Netdata (2020) *What is Netdata?* (Accessed on 25/4/2020). [Online], Available: https://learn.netdata.cloud/docs/agent/what-is-netdata/

[17] White T (2009) *Hadoop: the definitive guide,* 1st edn. O'Reilly, Sebastopol

[18] Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I (2010) *Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX conference on hot topics in cloud computing*. pp 10.

[19] Köster, Johannes, and Sven Rahmann. (2012) *Snakemake—a scalable bioinformatics workflow engine*, Bioinformatics 28.19: 2520-2522.

[20] Merkel D. (2014) *Docker: Lightweight Linux Containers for Consistent Development and Deployment.* (Accessed on 25/4/2020). [Online], Available:

https://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment
[21] Flask (2020) *Welcome to Flask*. (Accessed on 17/4/2020). Available: https://flask.palletsprojects.com/
[22] Nunes T., Campos D., Matos S., Oliveira J. L. (2013). *BeCAS: biomedical concept recognition services and visualization.* Bioinformatics, 29(15), 1915-1916.
[23] Doğan, Rezarta I., Leaman R., and L. Zhiyong. (2014) *NCBI disease corpus: a resource for disease name recognition and concept normalization.* Journal of biomedical informatics 47: 1-10
[24] Kurtzer G.M., Sochat V., Bauer M.W. (2017) *Singularity: Scientific Containers for mobility of compute*. PLoS ONE 12: e0177459.
[25] Canon S., Jacobsen D. (2016) *Shifter: Containers for HPC. Cray User Group*.
[26] da Veiga Leprevost F., Gruning B.A., Alves Aflitos S., Röst H.L., Uszkoreit J., et al. (2017) *BioContainers: an open-source and community-driven framework for software standardization.* Bioinformatics 33: 2580-2582.
[27] Yuping X., Yongzhao Z. (2012), *Virtualization and cloud computing*. In Ying Zhang, editor, Future Wireless Networks and Information Systems, pages 305– 312, Berlin, Heidelberg, Springer Berlin Heidelberg.
[28] Bauer E., Adams R. (2012). *Reliability and Availability of Cloud Computing.* Wiley-IEEE Press, 1st edition.
[29] Couvares P., Kosar T., Roy A., Weber J., Wenger K. (2007). *Workflow management in condor*. In I. J. Taylor, E. Deelman, D. B. Gannon, & M. Shields (Eds.), Workflows for e-science. (p. 357-375). Springer London.
[30] Zhang, D., Yan, B.-H., Feng, Z., Zhang, C., & Wang, Y.-X. (2017). *Container oriented job scheduling using linear programming model. 2017 3rd International Conference on Information Management (ICIM).* doi:10.1109/infoman.2017.7950370
[31] Lin S. C., Fei L. X. et al. (2009). *A reference architecture for Scientific workflow management systems and the VIEW SOA solution*, IEEE Transactions on Services Computing, vol. 2, no. 1, pp. 79–92.
[32] Zhao J., Gomez-Perez J. M., Belhajjame K., Klyne G., Garcia-Cuesta E., Garrido A., Goble C. (2012). *Why workflows break — Understanding and combating decay in Taverna workflows. 2012 IEEE 8th International Conference on E-Science.* doi:10.1109/escience.2012.6404482
[33] Goble, C. A., Bhagat, J., Aleksejevs, S., Cruickshank, D., Michaelides, D., Newman, D., De Roure D. (2010). *myExperiment: a repository and social network for the sharing of bioinformatics workflows. Nucleic Acids Research, 38(suppl_2), W677–W682.* doi:10.1093/nar/gkq429
[34] Missier, P., Woodman, S., Hiden, H., & Watson, P. (2013). *Provenance and data differencing for workflow reproducibility analysis. Concurrency and Computation: Practice and Experience, 28(4), 995–1015.* doi:10.1002/cpe.3035
[35] Deelman, E., Gannon, D., Shields, M., & Taylor, I. (2009). *Workflows and e-Science: An overview of workflow system features and capabilities. Future Generation Computer Systems, 25(5), 528–540.* doi: 10.1016/j.future.2008.06.012
[36] Kanterakis, A., Potamias, G., Swertz, M. A., & Patrinos, G. P. (2018). *Creating Transparent and Reproducible Pipelines: Best Practices for Tools, Data, and Workflow Management Systems. Human Genome Informatics, 15–43.* doi:10.1016/b978-0-12-809414-3.00002-4
[37] Amstutz P., Andeer R., Chapman B., Chilton J., Crusoe M.R., Guimerá R.V., Hernandez G.C., et. al. (2016). Common workflow language, draft 3, Unpublished Paper, Specifications.

[38] Perkel, J. M. (2019). *Workflow systems turn raw data into scientific knowledge. Nature, 573(7772), 149–150.* doi:10.1038/d41586-019-02619-z
[39] Bernstein D., (2014). *Containers and cloud: From lxc to docker to Kubernetes*, IEEE Cloud computing, vol. 1, pp. 81-84, 2014.
[40] Silva, L. M., Alonso, J., & Torres, J. (2009). *Using Virtualization to Improve Software Rejuvenation.* IEEE Transactions on Computers, 58(11), 1525–1538.
[41] Foster I. and Kesselman C. (1998). *Globus: A Metacomputing Infrastructure Toolkit, International Journal of Supercomputer Applications*, 11 (2). 115-129, 1988
[42] Torque (2020) *General*. (Accessed on 4/7/2020). Available: https://hpc-wiki.info/hpc/Torque
[43] Fei X., & Lu S. (2012). *A Dataflow-Based Scientific Workflow Composition Framework. IEEE Transactions on Services Computing, 5(1), 45–58.* doi:10.1109/tsc.2010.58
[44] Marchetti-Spaccamela A., Megow N., Schloter J., Skutella M., Stougie L. (2020). *On the Complexity of Conditional DAG Scheduling in Multiprocessor Systems*, Unpublished Paper.
[45] Melani, A., Bertogna, M., Bonifaci, V., Marchetti-Spaccamela, A., & Buttazzo, G. C. (2015). *Response-Time Analysis of Conditional DAG Tasks in Multiprocessor Systems. 2015 27th Euromicro Conference on Real-Time Systems.* doi:10.1109/ecrts.2015.26/