



# DEEPHEALTH

## D2.6 EDDL adaptation to cloud environments

<b>Project ref. no.</b>	<b>H2020-ICT-11-2018-2019 GA No. 825111</b>
<b>Project title</b>	Deep-Learning and HPC to Boost Biomedical Applications for Health
<b>Duration of the project</b>	1-01-2019 – 31-12-2021 (36 months)
<b>WP/Task:</b>	WP2/T2.4
<b>Dissemination level:</b>	PUBLIC
<b>Document due Date:</b>	31/05/2020 (M17)
<b>Actual date of delivery</b>	29/05/2020 (M17)
<b>Leader of this deliverable</b>	CRS4
<b>Author (s)</b>	Luca Pireddu (CRS4), Tatiana Silva (TREE), Maria A. Serrano, Eduardo Quiñones (BSC), Barbara Cantalupo (UNITO), Marina Zapater (EPFL), David Gonzalez (TREE), Marco Enrico Piras (CRS4), Tomas Teijeiro (EPFL)
<b>Version</b>	v7.0



This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 825111

## Document history

Version	Date	Document history/approvals
1	14/04/2020	Outline proposal
2	29/04/2020	Collected first draft of contents from TREE, CRS4, BSC, EPFL, UNITO
3	08/05/2020	Complete draft version (few comments outstanding)
4	14/05/2020	Internal review by Thales
5	18/05/2020	Complete version of the deliverable
6	25/05/2020	Review by the technical manager (Jon Ander Gómez) and PC (Monica Caballero)
7	28/05/2020	Final version

### DISCLAIMER

This document reflects only the author's views and the European Community is not responsible for any use that may be made of the information it contains.

### Copyright

© Copyright 2019 the DEEPHEALTH Consortium

This work is licensed under the Creative Commons License "BY-NC-SA".



# Table of contents

<b>DOCUMENT HISTORY</b>	<b>2</b>
<b>TABLE OF CONTENTS</b>	<b>3</b>
<b>1 EXECUTIVE SUMMARY</b>	<b>4</b>
<b>2 INTRODUCTION</b>	<b>4</b>
2.1 RELATION TO D3.6 - ECVL ADAPTATION TO CLOUD ENVIRONMENTS	5
2.2 BRIEF KUBERNETES BACKGROUND	6
2.3 DEEPHEALTH ON-PREMISE CLOUD	7
2.3.1 <i>Rancher</i>	8
2.3.2 <i>API Services</i>	8
<b>3 CONTAINER IMAGES AND CONTINUOUS INTEGRATION</b>	<b>23</b>
3.1 CONTAINER IMAGES	23
3.1.1 <i>Toolkit Images</i>	24
3.1.2 <i>Image Versioning</i>	25
3.2 IMAGE PUBLICATION ON DOCKERHUB	25
3.3 CONTINUOUS INTEGRATION PIPELINE	26
3.4 AVAILABILITY	27
<b>4 EDDL SUPPORT FOR CLOUD ENVIRONMENTS</b>	<b>28</b>
4.1 DISTRIBUTED EDDL OPERATIONS ON THE CLOUD	28
4.1.1 <i>Parallelization Approach</i>	28
4.1.2 <i>Supporting Kubernetes Cloud Environments with COMPSs</i>	28
4.1.3 <i>Preliminary Evaluation</i>	30
4.2 DEPLOYMENT OF FRONT-END FOR TRAINING MODELS ON THE CLOUD	31
<b>5 DEPLOYMENT-AS-A-SERVICE ON THE ODH PLATFORM</b>	<b>31</b>
5.1 ODH PLATFORM	31
5.2 THE STREAMFLOW FRAMEWORK	32
5.3 USING STREAMFLOW WITH THE DEEPHEALTH TOOLKIT	34
5.4 AN EDDL/ECVL PIPELINE EXAMPLE IN STREAMFLOW	36
5.4.1 <i>The Workflow</i>	37
5.4.2 <i>The Model and Bindings</i>	39
<b>6 OVERHEADS AND DRAWBACKS OF DEEP LEARNING ON THE CLOUD</b>	<b>40</b>
6.1 TEST CASE 1: MNIST PIPELINE	40
6.2 TEST CASE 2: EPILEPTIC SEIZURE DETECTION NETWORK	41
<b>7 CONCLUSIONS</b>	<b>43</b>

## 1 Executive Summary

---

This report for D2.6 presents the results from the activities of T2.4, *EDDLL Adaptation to Cloud Environments*. The goal of this task was to extend the EDDL library and related components of the DeepHealth toolkit to make it straightforward to use them on cloud computing resources, including scenarios featuring multi-cloud or hybrid HPC + cloud infrastructures. The importance of making DeepHealth compatible with cloud-native infrastructures, given the growth in adoption and availability of this type of resource, has been recognized since the inception of the project – in fact, the project includes this type of activity to target both the EDDL and the companion ECVL library (T3.3 - *ECVL Adaptation to cloud environments*).

A discussion around several important factors has taken the consortium to decide to target the Kubernetes container orchestrator as the DeepHealth cloud execution platform, instead of targeting bare infrastructure as a service. This decision was motivated by factors such as the need to avoid vendor lock-in due to incompatibilities between cloud services and the requirement to work with software containers. Thus, a full spectrum of solutions has been delivered to run the EDDL and the rest of the DeepHealth toolkit on the Kubernetes platform. At the lower level, Docker container images have been provided. At a higher level, the DeepHealth front end has been ported to the cloud and the DeepHealth libraries have been integrated into the ODH platform and the StreamFlow workflow manager, offering ready-to-use cloud-enabled solutions for expert users. From a scalability perspective, the EDDL has been extended to be able to efficiently exploit large-scale multi-cloud and hybrid cloud infrastructures, and cloud resources have been made available to consortium partners through the deployment of on-premise private cloud. Finally, continuous integration pipelines have been put in place to ensure that as the development of the DeepHealth libraries continues, those improvements will be automatically integrated into new container images so that the solutions described in this document remain up-to-date and sustainable in time.

Further, the advancement of the DeepHealth project has revealed it advantageous to adopt a solution where the EDDL and ECVL tightly interoperate within the DeepHealth toolkit. Thus, while the original project workplan structured the respective activities T2.4 and T3.3 as independent entities, in our implementation we have gone beyond the objective of enabling the use of each individual library on the cloud and have aimed for the goal of enabling the use of both DeepHealth libraries *together*, for the creation of complete cloud-enabled state-of-the-art deep learning pipelines.

A final note regarding the impact of the COVID-19 pandemic on the activities relevant to this report. Fortunately, the pandemic has only had minor effects on these activities, mostly in terms of slightly reduced productivity due to the total absence of face-to-face interaction between collaborators and also as individuals work to manage personal situations caused by the imposed restrictions (e.g., closed schools and daycares). Nevertheless, the consortium has still been able to effectively organize its efforts and deliver these results according to schedule.

## 2 Introduction

---

This deliverable reports on the outcomes of the activities in Task 2.4, which aimed to facilitate and demonstrate the use of the DeepHealth EDDL on cloud computing infrastructure. Since the inception of the DeepHealth project, facilitating the use of the DeepHealth toolkit on cloud infrastructure has been recognised as strategic. Cloud resources provisioned as a service are flexible, scalable, elastic, programmable and accessible with low up-front capital investment. Thus, the cloud is an important source of computing power for many usage scenarios - including deep learning applications.

In the DeepHealth project, the discussion around how to best support the use of the EDDL and the entire DeepHealth toolkit on cloud computing resources resulted in the decision to target the Kubernetes (k8s) container orchestrator as the cloud execution platform instead of targeting bare infrastructure as a service. This decision was motivated by several important factors. First, Kubernetes provides a platform that is agnostic to the underlying cloud provider. The growing adoption of cloud computing resources has motivated growing support for the main open cloud provisioning

solution (OpenStack<sup>1</sup>) and the entrance into the market of many commercial vendors. While similar in many ways, these solutions each provide their own flavour of cloud resources that are not compatible with each other. Therefore, software must typically be adapted to work with each one for which compatibility is desired. Targeting the k8s platform puts the DeepHealth toolkit on a level above these compatibility problems and makes it automatically usable with any common infrastructure as a service provider. In fact, k8s has become a de facto standard distributed container orchestration platform. It is already offered as a managed service by many cloud vendors, and for those users that need to deploy their own k8s cluster community-supported tools with support for different cloud providers already exist (e.g., KubeSpray<sup>2</sup>, Kops<sup>3</sup>). A second important reason for targeting Kubernetes is to allow the cloud adaptation of the DeepHealth toolkit to be based around software containers rather than virtual machines. Software containers have been demonstrated to be a modular, flexible and efficient approach to deploying software, and they can be used in both cloud and HPC scenarios. The Kubernetes platform, being a container orchestrator, treats containers as first-class citizens and thus greatly facilitates their use in complex scenarios. Finally, some of the DeepHealth use case platforms use Kubernetes or containers, so targeting these cloud technologies facilitates the uptake of the solutions created in these activities into those platforms and their related use cases.

Thus, the overarching goal of the DeepHealth cloud adaptation activities has been to enable the use of the DeepHealth toolkit on Kubernetes-based and hybrid Kubernetes-HPC computing infrastructures. The structure of this report mirrors the activities that have been carried out to achieve this goal, focusing on aspects particularly relevant to the EDDL. Specifically, Section 2.3 describes the on-premise Kubernetes cluster that has been deployed to ensure access to cloud resources to consortium members. Section 3 describes the container images that have been created for the various DeepHealth toolkit components and the continuous integration system that has been put in place to automatically generate new images as development of the EDDL, PyEDDL and other DeepHealth components continues throughout the project. In Section 4 we describe how PyEDDL has been extended to support distributed operation on Kubernetes clusters as well as simultaneously using Kubernetes and HPC computing resources. Next, Section 5 describes how we have provided DeepHealth functionality through the Streamflow framework on the Open DeepHealth (ODH) cloud platform, providing users with the means to declaratively define deep learning workflows that leverage the DeepHealth libraries in a platform as a service context. Finally, Section 6 provides an analysis of the efficiency costs paid for the adoption of a high-level containerized platform such as Kubernetes for performing a compute-intensive activity such as deep learning.

## 2.1 Relation to D3.6 - ECVL Adaptation to Cloud Environments

As an introductory note, it is important to highlight the relation between this report and the complementary report D3.6 *ECVL Adaptation to Cloud Environments*. While the original DeepHealth work plan structures the EDDL- and ECVL-related activities as distinct entities, advancement in the project has revealed it advantageous to adopt a solution where the EDDL and ECVL tightly interoperate within the DeepHealth toolkit. For instance, consider how any image-based DeepHealth model training or inference process performed by the EDDL is accompanied by image and dataset manipulation actions performed by the ECVL (e.g., dataset loading, splitting, image augmentation, etc.). Thus, it follows that a common concerted cloud adaptation effort for the entire DeepHealth toolkit was required from tasks T2.4 and T3.3 – rather than creating stand-alone solutions for each library. As a consequence, the results of the EDDL- and ECVL-specific tasks T2.4 and T3.3, which are respectively reported in this D2.6 and D3.6, are in many ways analogous as they solve the extended problem of facilitating the use of both the EDDL and the ECVL *together* on the cloud. In the interest of avoiding content duplication, when deemed appropriate these analogous results are described in

<sup>1</sup> Sefraoui, Omar, Mohammed Aissaoui, and Mohsine Eleuldj. "OpenStack: toward an open-source solution for cloud computing." *International Journal of Computer Applications* 55, no. 3 (2012): 38-42.

<sup>2</sup> <https://github.com/kubernetes-sigs/kubespray>

<sup>3</sup> <https://github.com/kubernetes/kops>

detail in only one of the two reports, while the other presents a summary. Specifically, in this report Section 4.2 presents a summary of the work done to adapt the DeepHealth front end for training models for use on cloud resources, while the full solution is described in D3.6.

## 2.2 Brief Kubernetes Background

Kubernetes is a distributed container and microservice platform that orchestrates computing, networking and storage infrastructure to support user workloads. Software containers have been demonstrated to provide a good way to bundle and deploy applications. However, as application complexity increases – e.g., complex multi-component software applications, multi-node clusters, etc. – running deployments becomes increasingly difficult. Kubernetes supports the automation of much of the work required to maintain and operate such complex services in a distributed environment. It orchestrates the deployment of containers across cluster nodes, matching heterogeneous resource requirements and availability. It provides self-healing, automatically restarting or moving workloads away from broken computing nodes. Moreover, it provides a standard platform that can be deployed on any modern computing infrastructure and thus enables portability across infrastructure providers.

A k8s cluster is composed of one or more master nodes and a set of worker nodes (see Figure 1). Nodes may be virtual or physical machines, depending on the deployment scenario. Multiple master nodes can be used to provide high availability and to scale to larger numbers of worker nodes (according to its documentation, k8s can scale up to about 5000 worker nodes before its performance begins to degrade). The Kubernetes master nodes run essential cluster services (e.g., controller manager, scheduler, etcd, API server) and form the cluster's *control plane*. On the other hand, worker nodes run the user-scheduled containerized workloads in units called *pods*. A pod is the smallest deployable object in k8s; it encapsulates one or more tightly coupled software containers (e.g., Docker) that share resources, including a single IP address. Kubernetes' main services communicate directly through the cluster network, while all pods are directly connected to a *virtual overlay network* which exists only within the cluster (see Figure 1). This design allows all pods within the k8s cluster to directly address each other without resorting to Network Address Translation (NAT) even with very large numbers of pods (k8s supports running up to 150,000 pods in the same cluster before performance can begin to suffer).

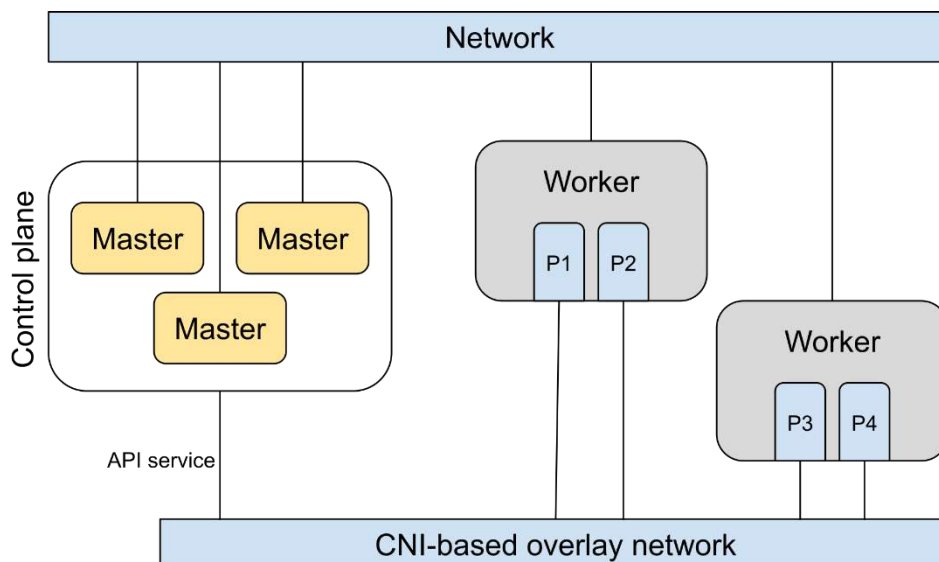


Figure 1 High-level Kubernetes cluster architecture. Boxes denoted as “Pn” are pods running on worker nodes.



To implement the overlay network Kubernetes we can use any of a number of plugins compatible with the Container Network Interface (CNI). As an example, consider Flannel<sup>4</sup>. Flannel allocates a subnet to each host participating in the overlay network from which pod IP addresses can be allocated<sup>5</sup>. Flannel agents run on each host in the Kubernetes cluster provide a TUN virtual network device for each pod in the Kubernetes cluster. In typical Flannel-based configuration, IP traffic between the pods on the same node is routed directly through a bridge network device managed by the container engine. On the other hand, traffic between pods on different hosts is encapsulated by the flannel agent on the pod's host and transmitted to the destination host through a standard Linux VXLAN – which encapsulates traffic in UDP datagrams – or potentially other mechanism (e.g., cloud-specific plugins); at the destination the corresponding Flannel agent interprets the packet and delivers it to the addressed overlay network device connected to the pod through the container engine's bridge.

## 2.3 DeepHealth On-Premise Cloud

As part of the DeepHealth activities, an on-premise *DeepHealth cloud*, consisting of a Kubernetes cluster, has been created to ensure access to cloud resources to consortium members; its architecture is shown in Figure 2. The cloud has been configured by TREE on their on-premise computing infrastructure. The computing resources provisioned for the cloud are flexible and can be modified as the project demands vary. It is built on commodity hardware and, as the project demands, GPU computing capabilities will be added – both through provisioning on the on-premise cluster and via nodes provisioned on vendor clouds (e.g., AWS). In fact, within the activities in Task 5.6 *Design and hybrid cloud computing solution to support DeepHealth libraries*, a hybrid platform will be constructed consisting of a Kubernetes cluster on-premise and another a second one running on Amazon Web Services (AWS) using Amazon Elastic Kubernetes Service (EKS) technology. At the time of this writing both use Kubernetes version 1.14.

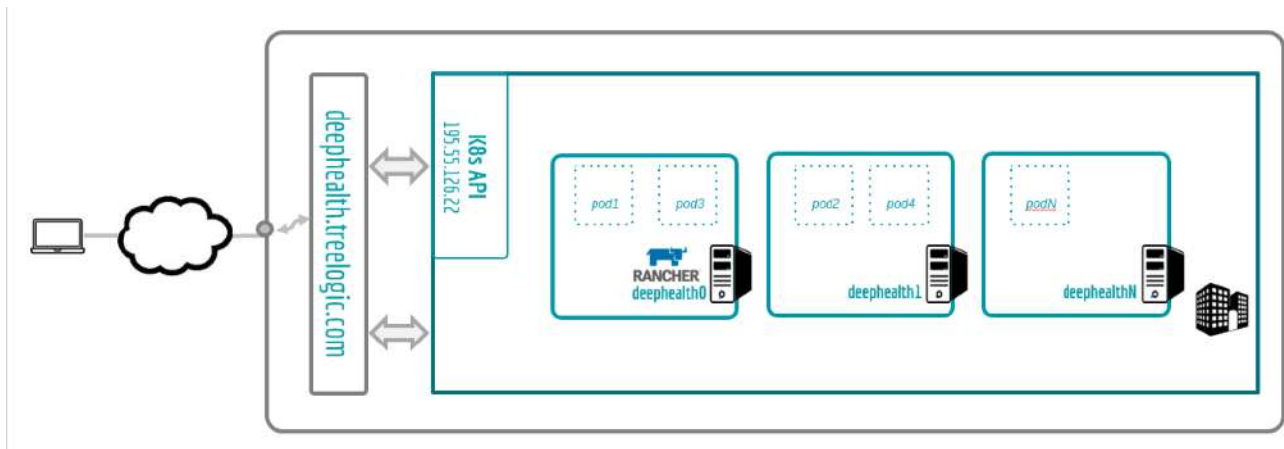


Figure 2 On-Premise cloud.

Furthermore, a high-level REST API has been developed on top of the k8s API to help abstracting the user from the infrastructure itself, simplifying the deployment and management of the workflows. It provides functions of varying complexity, from simple ones, like *listPods*, to more complex ones such as *exposePods* – which implements functionality abstracting the user from the potentially complex configuration of the clusters (e.g., multi-cloud, hybrid cloud, etc.). The API itself can support the addition of new k8s clusters both on-premise and in the cloud from any provider with the limitation of having a minimum k8s version of 1.14. To guarantee properly authenticated and authorized access to the DeepHealth cloud on TREE's infrastructure, connection through a VPN is required to access the API (which responds to the domain name deehealth.treeologic.com).

<sup>4</sup> <https://github.com/coreos/flannel>

<sup>5</sup> Marmol, Victor, Rohit Jnagal, and Tim Hockin. "Networking in containers and container clusters." *Proceedings of netdev 0.1* (2015).

Moreover, the DeepHealth cloud API supports the addition of new Kubernetes clusters to the platform using Azure Kubernetes Service (AKS) and Google Kubernetes Engine (GKE), as well as other on-premise clusters as long as they are all run the same version of k8s. The architecture of an example hybrid cloud environment composed of on-premise and AWS computing resources is illustrated in Figure 3.

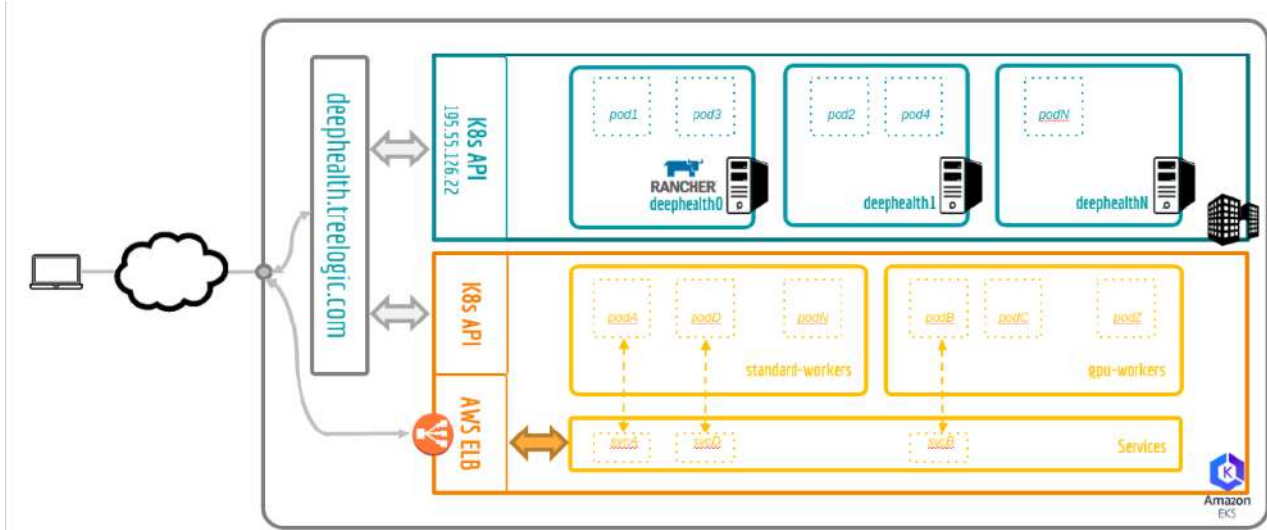


Figure 3 Hybrid Cloud Environment

### 2.3.1 Rancher

To support these clusters, an open-source Rancher multi-cluster orchestration platform was used<sup>6</sup>. Rancher allows managing the operational and security challenges on multiple Kubernetes clusters across any infrastructure. Its web interface provides control over deployments, jobs, pipelines, including an app catalog for fast deployment using the Helm software package manager<sup>7</sup>. Additionally, Rancher extends these best practices through automation and by making complex configurations easier to build.

Within a multi-cloud or hybrid-cloud context, a tool is needed to facilitate management and security tasks, as this can become a very error-prone and tedious task, while resources and Kubernetes clusters grow. Rancher meets these requirements, and after studying other similar tools (such as OpenShift), Rancher was chosen. With Rancher one can easily provision a new cluster in another provider and begin migrating workloads, all from within the same interface.

Rancher ships with tools for monitoring clusters, dashboards for visualizing metrics, an engine for generating alerts and sending notifications, a pipeline system to enable CI/CD for those not already using an external system. With a click it ships logs off to Elasticsearch, Kafka, Fluentd, Splunk, or syslog.

### 2.3.2 API Services

As previously mentioned, the DeepHealth cloud API was developed to facilitate the use and/or integration of the libraries with the Kubernetes platform. For its correct operation, the provisioned computing resources – by they in an on-premise cluster or an external cloud – must be described in its configuration. For users to access the API endpoint<sup>8</sup> provided by the DeepHealth cloud, VPN access is required to TREE's computing infrastructure where the service runs. A simple procedure

<sup>6</sup> <https://rancher.com/docs/rancher/v2.x/en/>

<sup>7</sup> <https://helm.sh/>

<sup>8</sup> <http://deephealth.treeologic.com/>



has been set up through which consortium members can request access. In this case, the API will act as a bridge between the DeepHealth cloud running on TREE and the COMPs (available in BSC), to deploy the EDDLl library in this cluster.

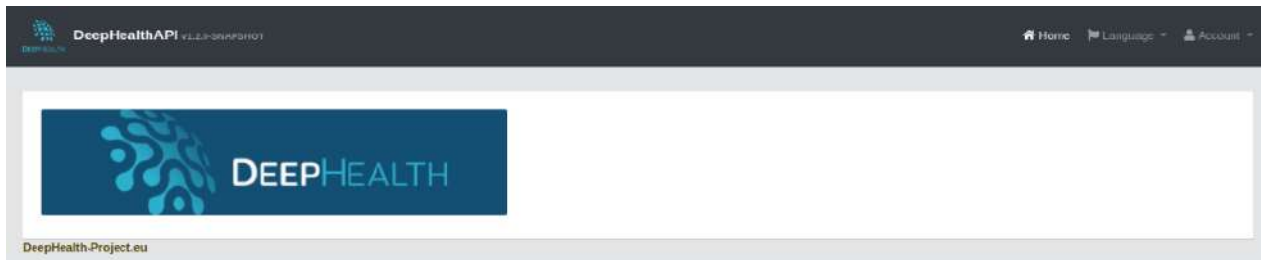


Figure 4 Once the VPN connection is established, the DeepHealth cloud API dashboard can be accessed.

Naturally, the API can be deployed in any environment. For that, a Docker image was created and is available in the DeepHealth's Docker-Hub repository<sup>9</sup>. In addition, this API has been defined with the OpenAPI specification<sup>10</sup>.

In the DeepHealth's GitHub, in the **deephealth-k8s** repository, a JSON file can be found, which can be used by REST clients to generate clients to be consumed<sup>11</sup>.

In Figure 5 the different methods implemented in the API are listed. The next sub-sections describe the different resources and verbs provided by the REST API and how to use them.

api-api-controller : the api API			Show/Hide	List Operations	Expand Operations
POST	/api/createServices/{namespace}/{pod}/{port}	createServices			
DELETE	/api/deleteDeployment/{namespace}/{deployment}	deleteDeployment			
DELETE	/api/deletePod/{namespace}/{pod}	deletePod			
GET	/api/describeServices/{namespace}/{service}	describeServices			
GET	/api/exposePod/{namespace}/{pod}/{port}	exposePod			
GET	/api/getLogs/{namespace}/{pod}	getLogs			
POST	/api/importYamlDeployments/{namespace}	importYamlDeployments			
POST	/api/importYamlPods/{namespace}	importYamlPods			
GET	/api/listDeployments/{namespace}	listDeployments			
GET	/api/listPods/{namespace}	listPods			
GET	/api/listPodsByDeployment/{namespace}/{deployment}	listPodsByDeployment			

Figure 5 Methods implemented by the API.

In order to work with Kubernetes' cluster of Azure or Google, slight changes must be made to the API. Within WP5 we will evaluate if it is necessary to include AKS and GKE technologies in the project. Currently, using AWS the needs of the project are covered.

<sup>9</sup> <https://hub.docker.com/r/dhealth/deephealth-api>

<sup>10</sup> <https://swagger.io/docs/specification/about/>

<sup>11</sup> <https://github.com/deephealthproject/deephealth-k8s/blob/master/openapi/api-docs.json>

### 2.3.2.1 Create Services

With this resource new services can be created. Input parameters are:

- **Namespace:** Namespaces give a scope for names. Resource names must be unique within a namespace, but not between namespaces. Namespaces cannot be nested within each other and each Kubernetes resource can only be found in one namespace.
- **Pod:** Name of the existing pod.
- **Port:** number of port to be exposed.
- **Cluster:** List of k8s clusters in which this method will be available. Options: currently only {aws}. In the coming months, {onpremise} will be available.
- **HTTP request:** `/api/createServices/{namespace}/{pod}/{port}`

**api-api-controller : the api API** Show/Hide List Operations Expand Operations

**POST** `/api/createServices/{namespace}/{pod}/{port}` createServices

**Response Class (Status 200)**  
ok

**Model** **Example Value**

```
{
  "message": "string",
  "service": "string"
}
```

**Response Content Type** application/json ▼

**Parameters**

Parameter	Value	Description	Parameter Type	Data Type
<b>cluster</b>	<span>aws ▼</span>	Cluster	query	string
<b>namespace</b>	<span>{required}</span>	Namespace	path	string
<b>pod</b>	<span>{required}</span>	Pod Name	path	string
<b>port</b>	<span>{required}</span>	Port	path	integer

**Response Messages**

HTTP Status Code	Reason	Response Model	Headers
201	Created		
401	Unauthorized		
403	Forbidden		
404	Not Found		

Try it out!

Figure 6 Create services method in the API.

The method will return the name of the service created and the following message codes:

- **CREATED:** If the service was created successfully.
- **EXISTS:** If the service was created previously.
- **ERROR:** If the service cannot be created (by incorrect pod name, or any other incorrect input parameter).

### 2.3.2.2 Delete Deployment

With this resource previously created deployments can be deleted. Input parameters are:

- **Namespace:** Namespaces give a scope for names. Resource names must be unique within a namespace, but not between namespaces. Namespaces cannot be nested within each other and each Kubernetes resource can only be found in one namespace.
- **Deployment:** Name of the deployment you want to delete.
- **Cluster:** List of k8s clusters in which this method will be available. Options: {onpremise} or {aws}.
- **HTTP request:** `/api/deleteDeployment/{namespace}/{deployment}`

DELETE
/api/deleteDeployment/{namespace}/{deployment}
deleteDeployment

Response Class (Status 200)

ok

Model | Example Value

```

{
  "message": "string"
}

```

Response Content Type application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
cluster	<span>onpremise</span>	Cluster	query	string
deployment	<span>(required)</span>	Deployment	path	string
namespace	<span>(required)</span>	Namespace	path	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
204	No Content		
400	error	Model   Example Value	

```

{
  "message": "string"
}

```

Figure 7Delete deployment method in the API.

The service returns an *OK* message if it has been successfully deleted, or in opposition, an *ERROR* message if it could not be deleted.

### 2.3.2.3 Delete Pod

Through this resource previously created Pods can be deleted. Input parameters:

- **Namespace:** Namespaces give a scope for names. Resource names must be unique within a namespace, but not between namespaces. Namespaces cannot be nested within each other and each Kubernetes resource can only be found in one namespace.
- **Pod:** Name of the pod you want to remove.
- **Cluster:** List of k8s clusters in which this method will be available. Options: {onpremise} or {aws}.
- **HTTP request:** `/api/deletePod/{namespace}/{pod}`

DELETE `/api/deletePod/{namespace}/{pod}` deletePod

**Response Class (Status 200)**  
ok

Model Example Value

```

{
  "message": "string"
}

```

Response Content Type application/json ▼

**Parameters**

Parameter	Value	Description	Parameter Type	Data Type
cluster	<span>onpremise ▼</span>	Cluster	query	string
namespace	<span>(required)</span>	Namespace	path	string
pod	<span>(required)</span>	Pod	path	string

**Response Messages**

HTTP Status Code	Reason	Response Model	Headers
204	No Content		
400	error	Model Example Value	

```

{
  "message": "string"
}

```

Figure 8 Delete Pod method in the API.

The service returns an *OK* message if the Pod has been successfully deleted, or conversely, an *ERROR* message if it could not be deleted.

### 2.3.2.4 Describe Services

With this resource we can describe the service created previously using the *createServices* method. Input parameters:

- **Namespace:** Name of space where the services are hosted.
- **Service:** Name of the service to be used. It usually consists of the svc prefix followed by the name of the pod (svc-podName).
- **Cluster:** List of k8s clusters in which this method will be available. Options: {onpremise} or {aws}.
- **HTTP request:** `/api/describeServices/{namespace}/{service}`

GET
/api/describeServices/{namespace}/{service}
describeServices

Response Class (Status 200)  
ok

Model | Example Value

```

{
  "external_ip": "string",
  "message": "string",
  "service": "string"
}

```

Response Content Type application/json ▼

Parameters

Parameter	Value	Description	Parameter Type	Data Type
cluster	aws ▼	Cluster	query	string
namespace	(required)	Namespace	path	string
service	(required)	Service	path	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

Try it out!

Figure 9 Describe services.

This service returns the message *OK* if everything has gone well, or on the contrary, *ERROR*, if the service failed, as well as the name of the service and the URL which will be available to use.

### 2.3.2.5 Expose Pod

This resource combines **createServices** and **describeServices** documented above. The resource creates and describes a service. **HTTP request** is `/api/exposePod/{namespace}/{pod}/{port}`

GET
/api/exposePod/{namespace}/{pod}/{port}
exposePod

Response Class (Status 200)

ok

Model
Example Value

```

{
  "external_ip": "string",
  "message": "string",
  "service": "string"
}

```

Response Content Type
application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
cluster	aws	Cluster	query	string
namespace	(required)	Namespace	path	string
pod	(required)	Pod Name	path	string
port	(required)	Port	path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

Try it out!

Figure 10 Expose Pod method, implemented in the API.

The answer will be the same as in the **describeService** method.

### 2.3.2.6 Get Pod's Log

Through this resource the log of a pod in the indicated namespace can be retrieved. Input parameters:

- **Namespace:** Name of space where the pods are hosted.
- **Pod:** Name of the pod from which the log will be obtained.
- **Lines:** Number of lines in the log (by default 100). Limits the number of lines obtained from the last line of the log to the first one.
- **Cluster:** List of k8s clusters in which this method will be available. Options: {onpremise} or {aws}.
- **HTTP request:** `/api/getLogs/{namespace}/{pod}`



Figure 11 Get Pod's method implemented.

### Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#) [Hide Response](#)

### Curl

```
curl -X GET --header 'Accept: text/plain' --header 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpZZWl0aWZhZS1mFjE'
```

### Request URL

```
http://localhost:8080/api/getLogs/pre-pre/name-pod-89c44f599-lfs8g?lines=10
```

### Response Body

```
[2020-02-25 07:38:58,867] INFO Server environment:os.arch=amd64 (org.apache.zookeeper.server.ZooKeeperServer)
[2020-02-25 07:38:58,867] INFO Server environment:os.version=3.10.0-1062.1.1.el7.x86_64 (org.apache.zookeeper.server.ZooKeeperServer)
[2020-02-25 07:38:58,867] INFO Server environment:user.name=root (org.apache.zookeeper.server.ZooKeeperServer)
[2020-02-25 07:38:58,867] INFO Server environment:user.home=/root (org.apache.zookeeper.server.ZooKeeperServer)
[2020-02-25 07:38:58,867] INFO Server environment:user.dir=/ (org.apache.zookeeper.server.ZooKeeperServer)
[2020-02-25 07:38:58,876] INFO tickTime set to 3000 (org.apache.zookeeper.server.ZooKeeperServer)
[2020-02-25 07:38:58,876] INFO minSessionTimeout set to -1 (org.apache.zookeeper.server.ZooKeeperServer)
[2020-02-25 07:38:58,876] INFO maxSessionTimeout set to -1 (org.apache.zookeeper.server.ZooKeeperServer)
```

Figure 12 Response of Get Pod's method.

### 2.3.2.7 Import Deployment's YAML

This resource implements the service declared in a YAML document provided as an argument. Input parameters are:

- **YAML:** this YAML file is of Deployment type (*kind: Deployment*)
- **Namespace:** Name of space where the deployment will be hosted.
- **Cluster:** List of k8s clusters in which this method will be available. Options: {onpremise} or {aws}.
- **HTTP request:** `/api/importYamlDeployments/{namespace}`

POST
/api/importYamlDeployments/{namespace}
importYamlDeployments

Response Class (Status 200)

ok

Model Example Value

```
{
  "internal_ips": {},
  "message": "string"
}
```

Response Content Type application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
body	(required)	Yaml Deployment	body	string
cluster	onpremise	Cluster	query	string
namespace	(required)	Namespace	path	string

Parameter content type: text/plain

Response Messages

HTTP Status Code	Reason	Response Model	Headers
201	Created		
400	error	Model Example Value	

```
{
  "internal_ips": {},
  "message": "string"
}
```

Figure 13 Import deployment method.

The call returns a JSON with an OK message if the operation was successful and the pods with the IP where they were executed.

Curl

```

curl -X POST --header 'Content-Type: text/plain' --header 'Accept: application/json' --header 'Authorization: Bearer ey
kind: Deployment \
metadata: \
  namespace: pre-pre \
  name: podname \
  labels: \
    app: confluent \
    confluent: zookeeper \
spec: \
  replicas: 2 \
  selector: \
    matchLabels: \
      app: confluent \
      confluent: zookeeper \
  template: \
    metadata: \
      namespace: pre-pre \
      labels: \
        app: confluent \
        confluent: zookeeper \
    spec: \
      containers: \
        - name: confluent-zk \
          image: confluentinc/cp-zookeeper:5.8.0 \
          env: \
            - name: ZOOKEEPER_CLIENT_PORT \
              value: "2181" \
          ports: \
            - containerPort: 2181 \
              protocol: TCP \
              http://195.55.126.74:8080/api/importYamlDeployments/pre-pre'

```

Request URL

```

http://195.55.126.74:8080/api/importYamlDeployments/pre-pre

```

Response Body

```

{
  "message": "OK",
  "internal_ips": {
    "podname-6b9ddf956d-f8f5v": "10.42.1.195",
    "podname-6b9ddf956d-jb9cm": "10.42.0.250"
  }
}

```

Response Code

```

200

```

Figure 14 Response from Import deployment method.

### 2.3.2.8 Import Pod's YAML

From a YAML document, the resource creates the pod with the image indicated in the corresponding file. Input parameters:

- **YAML:** this YAML declaration is of Pod type (*kind: Pod*)
- **Namespace:** Name of space where the pods will be hosted.
- **Cluster:** List of k8s clusters in which this method will be available. Options: {onpremise} or {aws}.
- **HTTP request:** `/api/importYamlPods/{namespace}`

POST
/api/importYamlPods/{namespace}
importYamlPods

Response Class (Status 200)

ok

Model
Example Value

```
{
  "internal_ip": "string",
  "message": "string"
}
```

Response Content Type
application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
body	(required)	Yaml Pod	body	string
cluster	onpremise	Cluster	query	string
namespace	(required)	Namespace	path	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
201	Created		
400	error	Model Example Value	

```
{
  "internal_ip": "string",
  "message": "string"
}
```

Figure 15 Import Pod's YAML method implemented in the API.

The call returns a JSON with an OK message if the operation was successful and the IP where the pod was created.

Curl

```
curl -X POST --header 'Content-Type: text/plain' --header 'Accept: application/json' --header 'Authorization: Bearer ey
apiVersion: v1 \
kind: Pod \
metadata: \
  name: nginx-pod \
spec: \
  containers: \
  - name: nginx \
    image: nginx:1.7.9 \
    ports: \
    - containerPort: 80 'http://195.55.126.74:8080/api/importYamlPods/pre-pre'
```

Request URL

http://195.55.126.74:8080/api/importYamlPods/pre-pre

Response Body

```
{
  "internal_ip": "10.42.0.251",
  "message": "Running"
}
```

Response Code

201

Figure 16 Response of Import Pod's YAML method

### 2.3.2.9 List Deployments

This resource returns all the deployments created in the corresponding namespace. Input parameters are:

- **Namespace:** Name of the space where the pods are hosted.
- **Cluster:** List of k8s clusters in which this method will be available. Options: {onpremise} or {aws}.
- **HTTP request:** `/api/listDeployments/{namespace}`

GET
/api/listDeployments/{namespace}
listDeployments

Response Class (Status 200)  
ok

Model Example Value

```
[
  {
    "availableReplicas": 0,
    "name": "string",
    "namespace": "string",
    "readyReplicas": 0,
    "replicas": 0,
    "updatedReplicas": 0
  }
]
```

Response Content Type application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
cluster	onpremise	Cluster	query	string
namespace	(required)	Namespace	path	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

Try it out!

Figure 17List deployments method implemented in the API.

The response will be the list of deployments (Figure 18)

```

Response Body

[
  {
    "name": "ng-6dd86d77d-c26wv",
    "state": "Running",
    "podIP": "192.168.45.59",
    "startTime": "2020-04-24T10:22:24Z",
    "restart": 0
  },
  {
    "name": "ng-6dd86d77d-qgdnk",
    "state": "Running",
    "podIP": "192.168.33.18",
    "startTime": "2020-04-24T10:22:24Z",
    "restart": 0
  }
]

```

Figure 18 Response of List of deployments method implemented.

### 2.3.2.10 List Pods

This resource returns all pods created in the name space indicated. Input parameters are:

- **Namespace:** Name of the space where the pods are hosted.
- **Cluster:** List of k8s clusters in which this method will be available. Options: {onpremise} or {aws}.
- **HTTP request:** `/api/listPods/{namespace}`

GET
/api/listPods/{namespace}
listPods

Response Class (Status 200)  
ok

Model Example Value

```

[
  {
    "name": "string",
    "podIP": "string",
    "restart": 0,
    "startTime": "string",
    "state": "string"
  }
]

```

Response Content Type application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
cluster	onpremise	Cluster	query	string
namespace	{required}	Namespace	path	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

Try it out!

Figure 19 List PODs method implemented.

The call returns a JSON with the corresponding POD list (Figure 20).



```

Curl
curl -X GET --header 'Accept: application/json' --header 'Authorization: Bearer eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJhZG1pbi'

Request URL
http://195.55.126.74:8080/api/listPods/pre-pre

Response Body
[
  {
    "name": "name-pod-89c44f599-hwjpf",
    "state": "Running",
    "podIP": "10.42.1.194",
    "startTime": "2020-02-25T07:38:49Z",
    "restart": 0
  },
  {
    "name": "name-pod-89c44f599-lfs8g",
    "state": "Running",
    "podIP": "10.42.0.249",
    "startTime": "2020-02-25T07:38:49Z",
    "restart": 0
  },
  {
    "name": "nginx-pod",
    "state": "Running",
    "podIP": "10.42.0.251",
    "startTime": "2020-02-25T10:15:36Z",
    "restart": 0
  }
]

Response Code
200

```

Figure 20 Response of List PODs method.

### 2.3.2.11 List PODs by Deployment

This resource returns all pods associated with the corresponding deployment. Input parameters are:

- **Namespace:** Name of the space where the pods are hosted.
- **Deployment:** Name of the deployment from which you want to extract your pods.
- **Cluster:** List of k8s clusters in which this method will be available. Options: {onpremise} or {aws}.
- **HTTP request:** `/api/listPodsByDeployment/{namespace}/{deployment}`

GET
/api/listPodsByDeployment/{namespace}/{deployment}
listPodsByDeployment

Response Class (Status 200)  
ok

Model Example Value

```
[
  {
    "name": "string",
    "podIP": "string",
    "restart": 0,
    "startTime": "string",
    "state": "string"
  }
]
```

Response Content Type application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
cluster	onpremise	Cluster	query	string
deployment	(required)	Deployment	path	string
namespace	(required)	Namespace	path	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

Try it out!

Figure 21 List PODs by deployment method.

The call returns a JSON with the corresponding POD list (Figure 22).

Response Body

```
[
  {
    "name": "ng-6dd86d77d-c26wv",
    "state": "Running",
    "podIP": "192.168.45.59",
    "startTime": "2020-04-24T10:22:24Z",
    "restart": 0
  },
  {
    "name": "ng-6dd86d77d-qgdnk",
    "state": "Running",
    "podIP": "192.168.33.18",
    "startTime": "2020-04-24T10:22:24Z",
    "restart": 0
  }
]
```

Figure 22 Response of POD list method.

### 3 Container Images and Continuous Integration

The first step in enabling the DeepHealth toolkit to leverage the Kubernetes platform was to create Docker container images for the DeepHealth components. Container images are the basic building blocks of Kubernetes software deployments. They are a snapshot of software with all its dependencies bundled with at least a partial runtime configuration. From a container image, a container can be executed, thus making the software operational.

For the cloud adaptation of the DeepHealth toolkit, a full set of Docker container images for the toolkit libraries and other components have been designed and implemented; further, an automation pipeline has been put in place to automatically keep them up-to-date with new releases of the DeepHealth software components. In this section we will describe the achievements in this direction.

#### 3.1 Container Images

In designing the structure of the DeepHealth container images, we sought to provide convenience for the user, by building feature-packed images that were ready-to-use for development or ad hoc applications, but also provide leaner images that were better suited to focused, production applications – where unnecessary complexity should be avoided. The resulting set of images and their interdependence is illustrated in Figure 23.

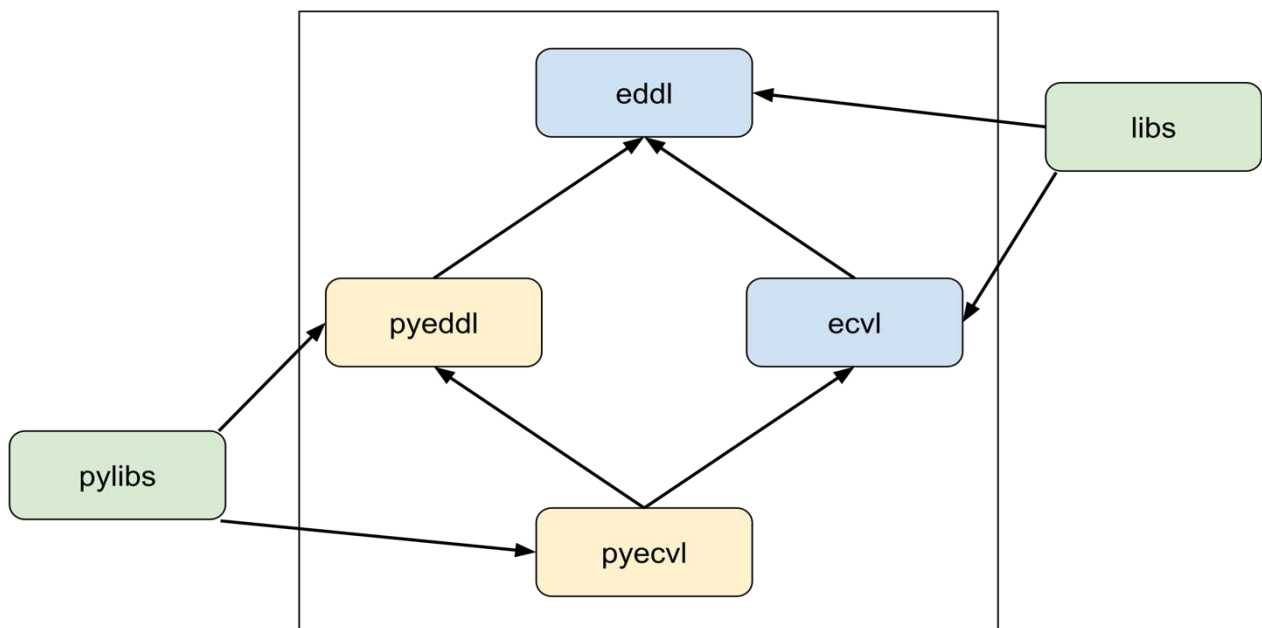


Figure 23 Relation between the main DeepHealth Docker images. C++ API libraries are shown in blue, Python API in yellow. The green images are conveniently package the pair of EDDL and ECVL libraries to support full pipelines.

Library-specific images are generated for each of the four toolkit libraries: EDDL, ECVL, PyEDDL and PyECVL. Further, full DeepHealth images are generated that package the entire C++ runtime (the `libs` image) and the entire Python runtime (the `pylibs` image). The latter two, which package the entire DeepHealth library functionality, are most easily used to integrate deep learning functionality into cloud-based applications. All these images contain all the DeepHealth *runtime requirements*, and are therefore ready to use to run DeepHealth-based applications that are grafted onto them. On the other hand, components not required for execution have been excluded to generate simpler and more compact images which have many advantages over larger images, including the following points:

- Shorter start-up time. Starting containers from more complex images can require more time as more data must be read from disk. This factor is amplified tremendously if the images are not cached yet and must be downloaded from the repository.

- Reduced disk usage. Smaller images require less disk space. This factor impacts particularly image repositories, which archive the images, but also the possibility to locally cache images and avoid the time-cost of downloading them repeatedly.
- Increased security. Removing components from the image reduces the attack surface where security problems can be introduced.
- Reduced complexity. By reducing the number of components included in the image, we reduce the number of ways things go wrong, and the effort required for developers to detect and fix bugs and security issues.
- Reduced build times. Smaller images are often quicker to build, which importantly affects the iteration time of the typical development cycle: code, build, test.

In general, beyond taking care to avoid automatic installation of components that are not strictly required (e.g., avoiding the installation of recommended packages), the image compilation processes all use a multi-stage build approach, which allows us to keep some things available during the build process but not include them in the final image.

**NVIDIA CUDA Support.** Given the nature and scale of the computations performed by the DeepHealth toolkit in practical applications, use of GPU hardware is a very high priority. The DeepHealth container images have been created to work seamlessly with the NVIDIA CUDA platform. The images are all built on the `nvidia/cuda:10.1-runtime` to include the CUDA runtime libraries and the configuration required to work with the NVIDIA Container Runtime for Docker. This component enables access to NVIDIA GPUs from within the software container with negligible overhead, thus enabling efficient GPU-accelerated computing in a containerized environment, such as Kubernetes.

### 3.1.1 Toolkit Images

The Docker images described thus far exclusively support the DeepHealth runtime requirements – i.e., they can be used to execute DeepHealth-based applications, but not to build or create them. We have created a set of *toolkit* Docker images for this purpose. In addition to the runtime library requirements, the toolkit images include the library compile time requirements (e.g., header files, compilers and other build tools, etc.) and are thus usable as a build environment for DeepHealth-based applications. In fact, they are also used for some of the build stages of the runtime images. For every DeepHealth runtime image there is a corresponding toolkit image, and these toolkit images mirror the relations between the regular images as shown in Figure 24.

As an example of the usage of a toolkit image, Figure 24 illustrates how a user can use the `eddl-toolkit` image to compile and run a program that uses the EDDL library without installing any software on the host computer except for Docker. This type of usage takes advantage of the fact that local directories from the host computer can be mounted into the container's file system, thus effectively being shared between the host and the container environments. In our example, the user could edit the program file normally, and switch to the containerized shell only to compile and execute it. Likewise, the toolkit images can also be used to work on the development of the DeepHealth libraries themselves.

```
$ docker run -it -u $(id -u) -v $(pwd):/my_examples --rm \
    dhealth/eddl-toolkit:latest /bin/bash

$ cd /my_examples/

$ g++ 6_mnist_auto_encoder.cpp -std=c++11 -leddl -pthread -o example

$ ./example
```

Figure 24 Full example showing how a toolkit image can be used to compile and run code residing on a host computer without locally installing any software other than Docker itself.

### 3.1.2 Image Versioning

The various images produced by DeepHealth have one of two main priorities: closely tracking the development of individual DeepHealth libraries or providing a combination of library snapshots that have been tested to work well together. The images `eddl`, `ecvl`, `pyeddl` and `pyecvl` are in the first category and track the progress of the corresponding DeepHealth library. For these, a new version of the image is automatically generated for every push to the corresponding library-specific software repository on GitHub by the DeepHealth continuous integration pipeline (described later in this document). The other images fall in the second category. For them, a new image version is generated when the DeepHealth developers manually tag a new release of the DeepHealth `docker-libs` repository<sup>12</sup>, which contains the code implementing the Docker-related DeepHealth library functionality.

*Table 1 Various DeepHealth images and the libraries they track. The library-specific images are automatically generated with each commit in the corresponding library source code repository; the `libs` and `pylibs` images generated by an automated pipeline that is manually triggered when developers decide to create a new release.*

Runtime image	Toolkit image	Library tracked	Dependencies included
dhealth/eddl	dhealth/eddl-toolkit	EDDLL	
dhealth/ecvl	dhealth/ecvl-toolkit	ECVL	EDDLL
dhealth/pyeddl	dhealth/pyeddl-toolkit	PyEDDLL	EDDLL
dhealth/pyecvl	dhealth/pyecvl-toolkit	PyECVL	PyEDDLL, EDDL and ECVL
dhealth/libs	dhealth/libs-toolkit	EDDLL+ECVL	
dhealth/pylibs	dhealth/pylibs-toolkit	PyECVL + PyEDDLL	EDDLL and ECVL

## 3.2 Image Publication on DockerHub

The Docker images produced by the DeepHealth project are published on DockerHub<sup>13</sup>. DockerHub is a Docker image hosting service, from which users and services can download images to execute containers or to extend them with additional functionality or customizations. A DeepHealth organization has been created<sup>14</sup> and it is the central access point for the official images produced by the project. In addition to the library and toolkit images, the project also publishes Docker images for the other specialized DeepHealth components.

<sup>12</sup> <https://github.com/deephealthproject/docker-libs>

<sup>13</sup> <https://hub.docker.com/>

<sup>14</sup> <https://hub.docker.com/orgs/dhealth>

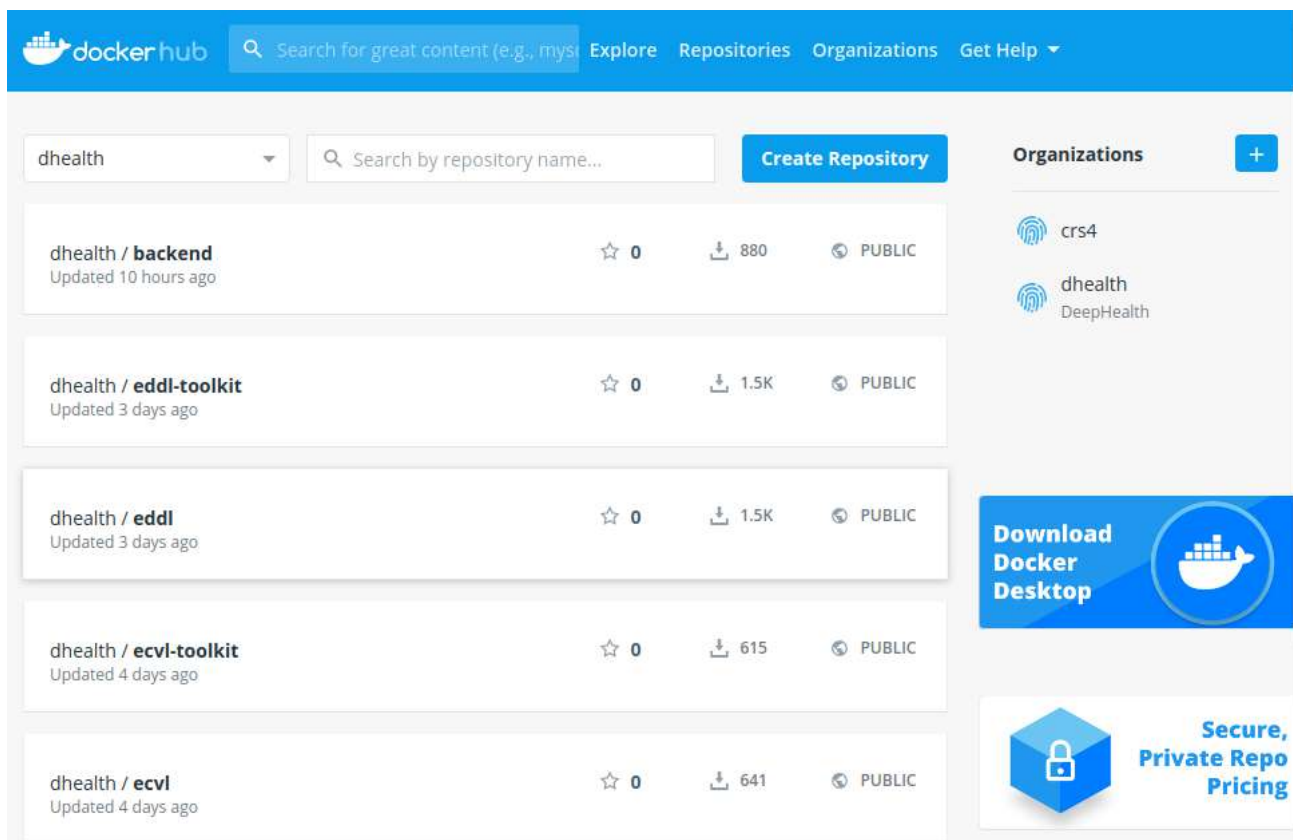


Figure 25 Screenshot of the DeepHealth organization on DockerHub with a partial list of its image repositories.

### 3.3 Continuous Integration Pipeline

In DeepHealth, continuous integration (CI) pipelines have been implemented to support the development process with continuous integration of new versions of DeepHealth software into the Docker images and the automated testing of those images. The full implementation is available in the [deephealthproject/docker-libs](https://github.com/deephealthproject/docker-libs) repository on GitHub<sup>15</sup>, runs on the Jenkins installation hosted by UNIMORE<sup>16</sup> and is installed next to the regular DeepHealth CI pipelines (on Jenkins, under the DeepHealth-Docker project folder instead of DeepHealth). As expected, the pipelines monitor the DeepHealth library repositories on GitHub. When a new change to the software is committed to a repository, the corresponding pipeline is activated and executes the following actions.

1. New Docker image(s) of the modified components is automatically compiled;
2. The tests for the component are automatically run. Unlike the CI pipelines for the bare software, which run the tests directly on a virtual machine, these CI pipelines run the tests within a Docker container – thus ensuring that the software works within a containerized environment.
3. If the tests pass, the new image is tagged appropriately and published on DockerHub. If the tests do not pass, a problem is reported on the Jenkins dashboard.

Various tags are applied to each published image to allow users to easily find and track the Dockerized version of the software they need. Specifically, the library images can be accessed by:

- `latest`: latest successful build of the image;
- `<commit id>`: latest image of the specific library git commit id;

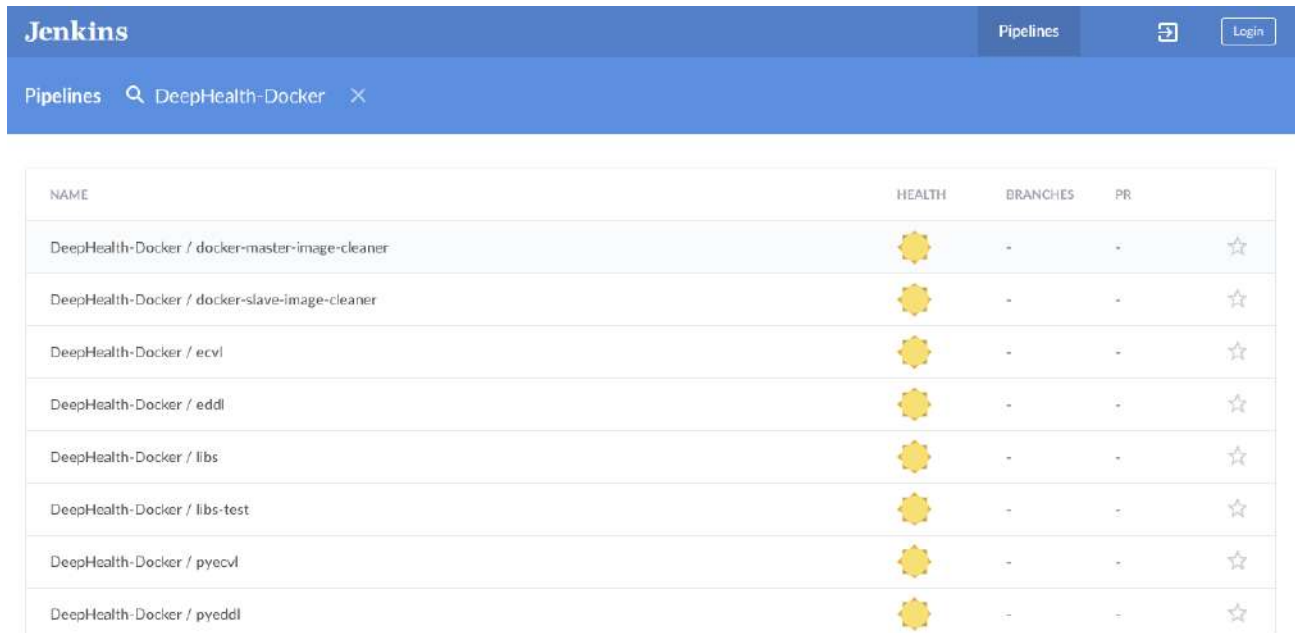
<sup>15</sup> <https://github.com/deephealthproject/docker-libs/>

<sup>16</sup> <https://jenkins-master-deephealth-unix01.ing.unimore.it>



- `<commit id>_<build id>`: image produced by the specific pipeline execution on the specific library git commit id.

This thorough tagging scheme allows us to keep all images that are published, ensuring reproducibility, while providing some simpler tags for convenience.











NAME	HEALTH	BRANCHES	PR	
DeepHealth-Docker / docker-master-image-cleaner		-	-	☆
DeepHealth-Docker / docker-slave-image-cleaner		-	-	☆
DeepHealth-Docker / ecvl		-	-	☆
DeepHealth-Docker / eddl		-	-	☆
DeepHealth-Docker / libs		-	-	☆
DeepHealth-Docker / libs-test		-	-	☆
DeepHealth-Docker / pyecvl		-	-	☆
DeepHealth-Docker / pyeddl		-	-	☆

Figure 26 The DeepHealth Jenkins dashboard showing the status of several pipelines.

As can be seen from the Jenkins dashboard, each DeepHealth Docker container image has its own specific CI pipeline. However, as much of the implementation as possible is shared among all of them. Specific parts adapt the different projects to the standard CI pipeline implementation. In particular, the various libraries have their specific requirements in terms of build systems (e.g., cmake, python setuptools), interfaces for the execution of tests, and provisioning of dependencies that must be handled by the pipeline. The core of the CI pipeline can also be downloaded and executed locally. This feature is particularly useful in the development of cloud-enabled DeepHealth components since new custom container images, perhaps including new DeepHealth code, can be built and tested locally.

### 3.4 Availability

All the code that implements the DeepHealth container images and the container CI pipelines described in this section has been released as open source and is available in the `deephealthproject/docker-libs` repository on GitHub. The DeepHealth Docker images are published in the publicly accessible image repositories under the “dhealth” organization on Docker Hub<sup>17</sup>.

<sup>17</sup> <https://hub.docker.com/orgs/dhealth>

## 4 EDDLl Support for Cloud Environments

### 4.1 Distributed EDDLl Operations on the Cloud

This section describes the distribution of the computation of the EDDLl training operations in the DeepHealth cloud environment (see Sec. 2.3). The work distribution strategy has actually been implemented through the PyEDDLl Python bindings for the EDDLl library, because these are more readily compatible with the COMPSs framework. We refer the interested reader to check deliverables D2.1 *EDDLl library* (May 2020) for a complete description of the parallelization strategy for the training operation, and D5.4 *The runtime system for DeepHealth libraries* (March 2020) for the modifications included in the COMPSs runtime to support the execution on in cloud environments.

Concretely, this section briefly describes the parallel structure of the PyEDDLl training operation and its distribution on the on-premise cloud and provides a first evaluation from a performance and accuracy point of view.

#### 4.1.1 Parallelization Approach

The parallelism exposed by the PyEDDLl training operation is shown in Figure 27 in the form of a task dependency graph (TDG). Each node of the TDG represents a COMPSs task that can be distributed among the different computing nodes – i.e., pods in the case of the DeepHealth cloud; the edges of the TDG represent the data transfers and synchronization between tasks, defining an execution order.

In the flow illustrated in Figure 27 the neural network model is built in parallel on each distributed computing resource (`build` COMPSs tasks). As soon as the model is defined in a resource, a `train_batch` operation included in a task can start. Then, the parallelisation strategy follows a synchronous training approach: each `train_batch` COMPSs task operates in parallel over a subset of the dataset (divided into a given number of batches). When all tasks complete, the computed (partial) weights are collected in the master node (`update_gradients` task). This process is repeated for a given number of epochs (see deliverable D2.1 *EDDLl library* for further details).

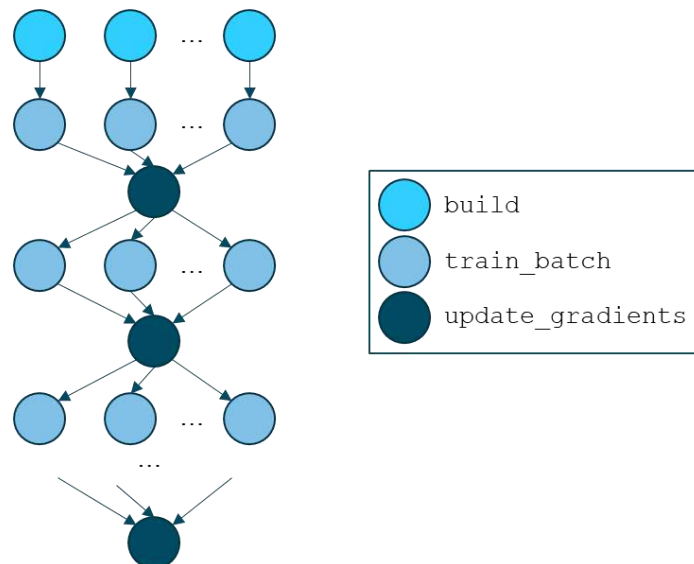


Figure 27 Task-level parallelism of the PyEDDLl training operations.

#### 4.1.2 Supporting Kubernetes Cloud Environments with COMPSs

One of the main features of the COMPSs framework is that it abstracts the parallel execution model from the underlying distributed infrastructure. Hence, COMPSs programs do not include any detail that would tie them to a particular platform, boosting portability among diverse infrastructures and so

enabling its execution in both a classical HPC environment and a cloud-based environment. COMPSs abstracts the underlining infrastructure by creating a set of execution environments, named *COMPSs workers*, in which COMPSs tasks execute. Internally, the COMPSs runtime implements different *adapters* to support the execution of COMPSs tasks in a given resource. Through a set of configuration files the user specifies the available sources of computing resources, which may reside in clusters or in cloud. Specifying a mixture of HPC and cloud resources naturally results in a hybrid cloud execution scenario, where the job is executed partly on HPC and partly on cloud resources.

Within the context of the cloud adaptation work in DeepHealth, we have extended the COMPSs runtime with a new adapter to support the specific DeepHealth cloud API services described in Section 2.3.2 (see deliverable D5.4 *The runtime system for DeepHealth libraries* for further information). This adapter allows COMPSs to provision workers running on Kubernetes through the API; thanks to the nature of the API, depending on the user-provided configuration these workers can be dispatched transparently on different clouds, creating a multi-cloud execution environment. However, this functionality has thus far only been tested with the on-premise DeepHealth cloud.

Concretely then, to run EDDL training on cloud resources the user configures the available provisioners of computing resources through *.xml* configuration files. As an example, Figure 28 presents a first draft of the information to be included to deploy and execute the distributed version of the PyEDDL training operation on the on-premise DeepHealth cloud infrastructure described in Section 2.

```
<ResourcesList>
  <CloudProvider Name="treeologic">
    <Endpoint>
      <Server>http://deephealth.treeologic.com</Server>
      <ConnectorJar>treeologic-conn.jar</ConnectorJar>
      <ConnectorClass>es.bsc.conn.treeologic.Treeologic</ConnectorClass>
    </Endpoint>
    <Properties>
      <Property>
        <Name>User</Name>
        <Value>XXXX</Value>
      </Property>
      <Property>
        <Name>Password</Name>
        <Value>XXXX</Value>
      </Property>
    </Properties>
    <Image Name="bscpc/compss-deephealth-demo">
    ...
```

Figure 28 Cloud information included in the COMPSs *.xml* configuration files.

The information included in the *.xml* files is the following:

- **<CloudProvider>**: Informs to COMPSs about the cloud provider used. If the “treeologic” cloud is selected, COMPSs will make use of the API services described in Section 2.3.2 to internally deploy the PyEDDL operations parallelised with COMPSs.
- **<Endpoint>**: Defines the connection details to the cloud infrastructure.
- **<Properties>**: Defines the access details, including username and password.
- **<Image>**: Defines the location of the docker image in which the parallelised version of the PyEDDL with COMPSs is located.

It is important to remark that the number of containers in which the image will be deployed (named *pod replicas* in Kubernetes) is not included in the *.xml* configuration files yet. Instead, the current implementation collects information from an environment variable. Future releases of the COMPSs runtime will include this information into the *.xml* files.

Figure 29 shows a possible distribution of the execution of the parallel version of the PyEDDLL training in the TREE DeepHealth cloud considering three COMPSs workers. This configuration has been defined with the parameters defined in Figure 28 and setting the number of replicas to 3.

In this initial integration phase of the project, the user is responsible for “manually” deploying and launching the execution of the COMPSs Master (deployed in the *Deephealth0* computing node in the figure) through the DeepHealth cloud API. The COMPSs runtime will then use also the DeepHealth cloud API to automatically deploy the rest of replicas (three in this case) in which the COMPSs workers will be executed. Once the deployment is completed, the parallel execution of the training operation is initiated, and so the COMPSs runtime starts the distribution the different COMPSs tasks (i.e., the *build* and *train\_batch* tasks shown in Figure 27, guaranteeing the data dependencies among tasks. In this case, the *update\_gradients* function is executed in the COMPSs master to aggregate the partial computed weights at the end of each epoch.

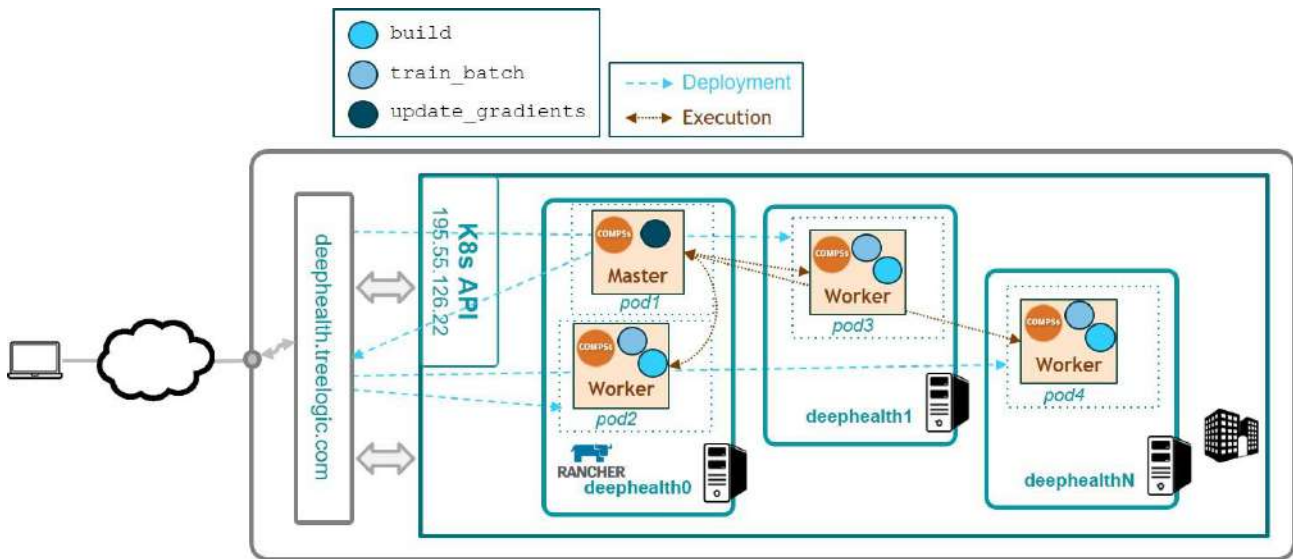


Figure 29 Distributed PyEDDLL training on the DeepHealth cloud.

### 4.1.3 Preliminary Evaluation

We have conducted a preliminary set of experiments in the DeepHealth cloud to evaluate the performance speedup of the distributed training operation when the trained accuracy is above 90%. Concretely, the experimental setup is the following:

- The MNIST training dataset<sup>18</sup>;
- A DNN topology (784 x 1024 x 1024 x 1024 x 10, linear rectified activation function for the hidden layers and softmax for the output layer);
- Kubernetes cluster on-premise featuring 3 nodes with the following characteristics:
  - *deephealth0* ---- 16Gb, 4vCPU(2x2) Intel(R) Xeon(R) CPU L5640 @ 3,00GHz
  - *deephealth1* ---- 8Gb, 4vCPU(2x2) Intel(R) Xeon(R) CPU X5570 @ 2,93GHz
  - *deephealth2* ---- 8Gb, 4vCPU(2x2) Intel(R) Xeon(R) CPU X5570 @ 2,93GHz;
- Number of COMPSs Workers = number of batches = 1 (sequential), 2, 3, 4, 8, 16.

Figure 30 Preliminary evaluation of the distributed PyEDDLL training in the cloud. Note that the COMPSs worker processes are distributed over the set of 3 computing nodes. This entails that from  $N > 4$  workers the COMPSs processes compete for resources on the same nodes, explaining the degrading performance at  $N = 16$ . shows the execution time and performance speedup of the distributed PyEDDLL training operation when varying the number of COMPSs workers (and so number of *pod replicas* deployed) from 1 to 16. As expected, the maximum performance is achieved

<sup>18</sup> <http://yann.lecun.com/exdb/mnist/>

when the number of replicas equals to the number of available computing nodes, i.e., 3. From this point on, the performance is degraded due to the execution interfaces when executing more than one COMPSs worker in the same node, thus adding scheduling and coordination overhead without adding computing resources.

The accuracy achieved in all experiments is 94.01%, independently of the distribution strategy.

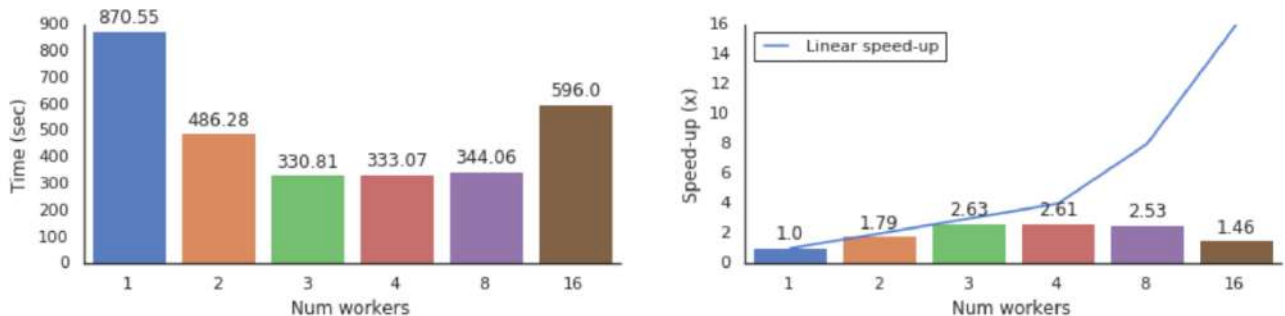


Figure 30 Preliminary evaluation of the distributed PyEDDLL training in the cloud. Note that the COMPSs worker processes are distributed over the set of 3 computing nodes. This entails that from  $N > 4$  workers the COMPSs processes compete for resources on the same nodes, explaining the degrading performance at  $N = 16$ .

## 4.2 Deployment of Front.End for Training models on the Cloud

The DeepHealth toolkit includes a high-level web service and a graphical web-based user interface to enable high-level access to the functionality offered by the DeepHealth libraries – described in D2.5 *EDDLL Toolkit front-end* and D3.5 *ECVL Toolkit front-end*. Together, these components are the DeepHealth front-end, which allows expert users to train new models or use existing models to perform inference without writing any programming code. As explained in those previous reports, given the tight interoperability between the EDDL and ECVL the implemented solution has unified the EDDL and ECVL front-ends into a single software package. As a result, the cloud adaptation of the front-ends for EDDL and ECVL has also been unified. The resulting work is described in detail in D3.6 *ECVL adaptation to cloud environments*.

## 5 Deployment-as-a-Service on the ODH Platform

As discussed in the Introduction (Section 2), Kubernetes is currently the reference platform for deploying, scaling and managing containerized applications, and Docker containers are an effective solution adopted to provide users with access to the images of the DeepHealth toolkit libraries (see Section 3.1). In this section, we present the StreamFlow framework, which has been developed for managing the deployment and the execution of tasks in multi-container environments, using different underlying technologies, such as Kubernetes, and supporting concurrent execution of communicating tasks. This framework also exploits the portability properties of software containers to simplify the execution of distributed applications based on DeepHealth libraries on different and possibly hybrid infrastructures. We also describe how StreamFlow works on the OpenDeepHealth (ODH) platform, thus demonstrating how the cloud adaptation activities have enabled the use of the DeepHealth toolkit not just on general-purpose clouds but also on more customized configurations.

### 5.1 ODH Platform

The OpenDeepHealth platform, designed to implement the DeepHealth project requirements, has been developed as part of the University of Turin's HPC4AI<sup>19</sup> infrastructure, which is a federated OpenStack<sup>20</sup> cloud with multi-tenant private Kubernetes instances. The ODH platform (see Figure 31)

<sup>19</sup> Marco Aldinucci et al. HPC4AI, an AI-on-demand federated platform endeavour. ACM Computing Frontiers 2018, Ischia, Italy, 8-10 May 2018. doi: 10.1145/3203217.3205340

<sup>20</sup> O. Sefraoui et al, "OpenStack: toward an open-source solution for cloud computing", Int. Journal of Computer Applications, vol 55, num3, 2012



is defined as an HPC secure tenant where a multi-tenant Kubernetes cluster is deployed and the DeepHealth toolkit libraries are available as Docker containers, both for CPU and GPU nodes. Technical details about the overall HPC4AI infrastructure and the ODH configuration can be found in deliverables D4.1 Integration of DeepHealth platforms and use cases and D5.1 Efficient HPC Infrastructure for DeepHealth libraries.

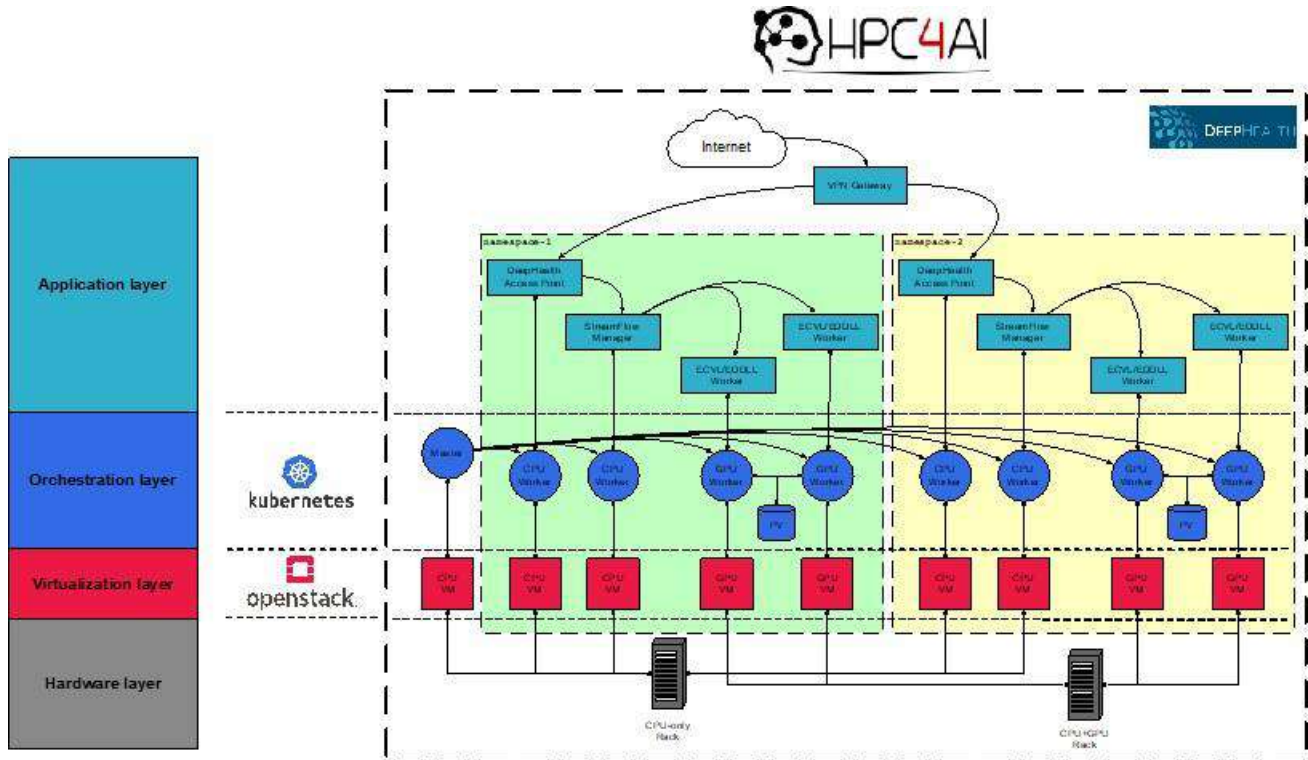


Figure 31 OpenDeepHealth platform.

## 5.2 The StreamFlow Framework

In addition to the Kubernetes cluster, at a higher level of abstraction, UNITO is developing StreamFlow<sup>21</sup>, a novel approach to workflow execution that supports sophisticated execution plans. In fact, StreamFlow allows the declarative description of possibly many execution environments to be used simultaneously for the execution of workflow nodes; properties associated to the workflows nodes are used to decide in which execution environment to schedule it. Thus, StreamFlow supports the deployment of an application on top of a "virtual" cross-site platform, making it possible to partition the application workflow and describe an execution plan spawning across multiple sites, even if they do not share the same data space. It also naturally handles workflows with nodes that have specific requirements, perhaps only available as a subset of the sites (e.g., particularly relevant to DeepHealth would be GPU nodes). It is worth noting that the ability to model an application as a workflow provides an interface between the domain specialists, specifying their own application requirements, and the computing infrastructure. The StreamFlow workflow manager implements an orchestration layer that allows users to easily manage workflow modelling, enhancing application reproducibility and portability, and to support deployment and execution in different environments, enhancing Kubernetes' ability to handle different computational steps.

The idea behind this approach is that the ability to deal with hybrid workflows – i.e., consisting of tasks running in different execution environments – can be a crucial aspect for performance optimization when working with massive amounts of input data and varying requirements in computational steps. Accelerators like GPUs, and in turn, different infrastructures like HPC and clouds, can be more

<sup>21</sup> StreamFlow: cross-breeding cloud with HPC Published in ArXiv 2020 <https://arxiv.org/abs/2002.01558>



efficiently used by selecting for each application the execution plan that better fits the specific needs of the computational steps. This final aspect is particularly relevant for the computationally and data-intensive applications developed in the DeepHealth project.

The StreamFlow framework is a container-native Workflow Management System (WMS) written in Python 3. It has been designed around two main principles:

- Allow the execution of tasks in multi-container environments in order to support concurrent execution of multiple communicating tasks in a multi-agent ecosystem;
- Relax the requirement of a single shared data space, in order to allow for workflow executions on top of multi-cloud or hybrid cloud/HPC infrastructures.

The basic idea behind the StreamFlow model is to provide the ability to express correspondences between the application workflow description, specifying application steps with the related data dependency, and an environment model description defining target infrastructures capability. This information allows StreamFlow to effectively orchestrate the deployment and execution of the application on the target infrastructure.

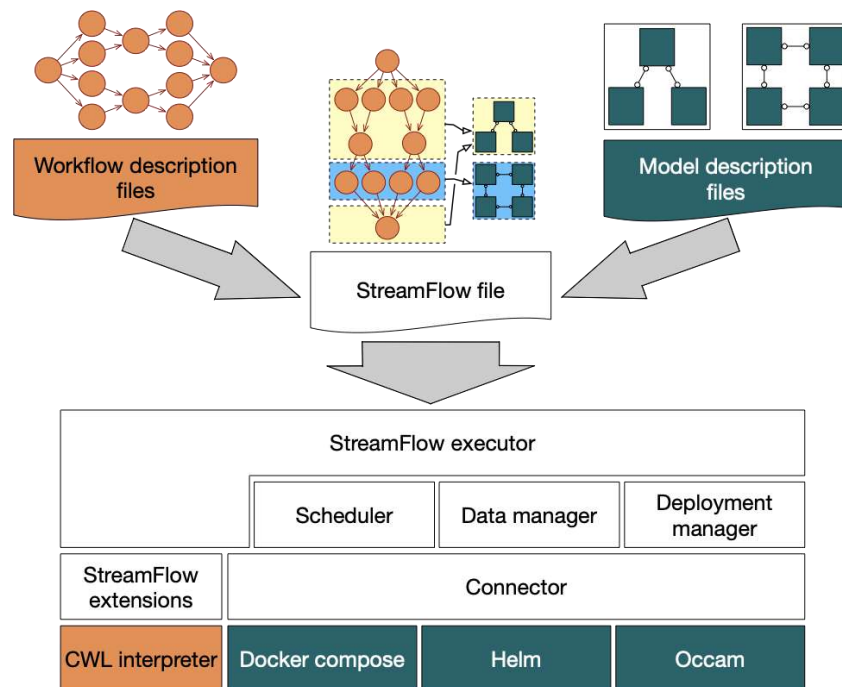


Figure 32 StreamFlow framework's logical stack. Colored portions refer to existing technologies, while white ones are directly part of StreamFlow. In particular, the orange area is related to the definition of the workflow's dependency graph, while the green area refers to the execution environments.

One of the design choices for the StreamFlow approach is to rely on existing coordination languages, instead of coming with yet another way to describe workflow models. Therefore, we decided to integrate with the CWL format because, being CWL a fully declarative language, it is far simpler to understand than its Make-like or dataflow-oriented alternatives. Nevertheless, since we plan to support other coordination languages in the future, a more agnostic mid-layer representation of a workflow graph is definitely in the next implementation steps list.

In Figure 32 the StreamFlow logical stack is depicted. In StreamFlow's glossary, a complex multi-container environment is called **model**. Each model is managed independently of the others by a dedicated **connector** implementation, which acts as a proxy for the underlying orchestration library (for instance, the Helm connector enables interaction with Kubernetes platforms). A single model can include multiple types of containers, called **services**. For example, a Docker Compose file describing a database and a Tomcat containers linked together constitute a model with two services.

In the StreamFlow architecture, the Deployment Manager has the role of creating models when needed and destroying them as soon as they are no longer required. It relies on the underlying orchestration library by means of pluggable implementation of the Connector interface. So far three different Connector implementations are included with StreamFlow: Docker Compose, Helm and OCCAM (Open Computing Cluster for Advanced data Manipulation)<sup>22</sup> – the HPC component of the UNITO platform. The Scheduler is in charge of selecting the best resource on which each task should be executed, while guaranteeing that all requirements are satisfied. Finally, the Data Manager, which knows where each task's input and output data reside, must ensure that each service has access to all the data dependencies required to complete the assigned task, performing data transfers only when necessary. At this point, a job, (i.e., the run-time representation of a task) can be successfully executed on the selected resource.

### 5.3 Using StreamFlow with the DeepHealth Toolkit

When launching StreamFlow execution, the only argument it accepts is the path of a YAML file, conventionally called `streamflow.yml`. This file serves to define the workflows to be executed, specify various parameters and, crucially, to link each task of the workflows with the service that should execute it. In order to make this binding unambiguous, each service in a model and each task in a workflow should be uniquely identifiable. The `streamflow.yml` file of the pipeline that will be presented in the next section is provided in Figure 33.

The **workflows** section consists of a dictionary with uniquely named workflows to be executed in the current run. Each workflow specification contains three fields:

- `type` that identifies which language has been used to describe the workflow dependency graph (at the moment `cwl` is the only accepted value),
- `config` that includes the paths to the files containing the graph description,
- a `bindings` list that contains the task-model associations.

Different workflows are totally independent of each other and this means that, even if two tasks in two different workflows can refer to the same model specification, two different environments will be actually deployed for their execution. Considering workflows as dependency graphs, each node can refer to either a simple task or a nested sub-workflow. Therefore, we decided to adopt a file-system based mapping of each task to a POSIX-like path, where each simple task is mapped to a file and each sub-workflow is mapped to a folder, which can contain both files and sub-folders.

The **models** section contains a dictionary of uniquely named model specifications, each of which is an object with two distinct fields:

- `type` that identifies which Connector implementation should be used for its creation, destruction and management;
- `config` which contains a dictionary with configuration parameters for the corresponding Connector. Usually, the config parameters are directly extracted from the tools commonly used to interact with the underlying orchestration library (e.g. helm CLI for Helm charts), so that a user who is familiar with these libraries can easily understand the StreamFlow format.

Finally, the format adopted for the **bindings** list takes into account all considerations on unambiguous identification of tasks and services. In particular, each element of this list contains:

- a `target` object, with a `model` and a `service` attributes that uniquely identify a service (in the example in Figure 33 the service is `pylibs`, a Python DeepHealth library container)
- a `step` attribute containing a path in the file-system abstraction of a workflow graph. If the path resolves to a folder (i.e., to a nested sub-workflow), the same target service is applied

---

<sup>22</sup> M. Aldinucci, et al., “The Open Computing Cluster for Advanced data Manipulation (OCCAM),” in Journal of Physics: Conf. Series 898 (CHEP 2016), San Francisco, USA, 2017.

recursively in the file-system hierarchy, unless a more specific configuration (i.e., another entry in the bindings list with a deeper path in its step overrides it.

The best way to unambiguously identify a service in a model strictly depends on the model specification itself. In Kubernetes (and consequently in Helm) the unit of deployment is a Pod, which can contain multiple containers inside it. In this case, the user is required to explicitly add the name attribute for each container in the Pod template and to ensure the uniqueness of such name in the context of the whole Helm release.

```
#!/usr/bin/env streamflow
version: v1.0
workflows:
  master:
    type: cwl
    config:
      file: cwl/main.cwl
      settings: cwl/config.yml
    bindings:
      - step: /
        deployments:
          - helm-odh-demonstrator
        target:
          model: helm-odh-demonstrator
          service: pylibs

models:
  helm-odh-demonstrator:
    type: helm
    config:
      chart: environment/helm/odh-demonstrator
      inCluster: true
      releaseName: use-case-pipeline
```

Figure 33 streamflow.yml (example)

A list of Docker containers supporting ML application development and execution is deployed in ODH platform (see Figure 34), including DeepHealth toolkit and libraries. These containers can be referred to as **services** in the StreamFlow models to deploy and execute any application that is integrated with them. Therefore, any application or tasks that is integrated in DeepHealth library containers can be easily deployed and executed on the ODH platform. As further step, we are investigating the possibility of integrating DeepHealth backend and frontend (see Section 4.2 and D3.6) in ODH and StreamFlow framework to provide complete access to the DeepHealth toolkit.






 <b>libs-toolkit</b>	Added DeepHealth containers
 <b>libs</b>	Added DeepHealth containers
 <b>pylibs</b>	Added DeepHealth containers
 <b>pytorch</b>	Added containers to opendeephealth repository
 <b>tensorflow</b>	Added containers to opendeephealth repository

Figure 34 ML containers in ODH

Besides being currently installed in ODH, StreamFlow it is also available on GitHub<sup>23</sup> and PyPI<sup>24</sup>. It can be executed directly from the command line with the following command:

```
streamflow /path/to/streamflow.yml
```

Docker images are also available on Docker Hub<sup>25</sup>. In order to run a workflow using an image the following steps are required.

- A StreamFlow project, containing a `streamflow.yml` file and all the other relevant dependencies (e.g., a CWL description of the workflow steps and a Helm description of the execution environment) needs to be mounted as a volume inside the container, for example in the `/streamflow/project` folder.
- Workflow outputs, if any, will be stored in the `/streamflow/results` folder. Therefore, it is necessary to mount the output path as a volume to persist the results.
- StreamFlow will save all its temporary files inside the `/tmp/streamflow` location. For debugging purposes, or in order to improve I/O performances in case of huge files, it could be useful to also mount this location as a volume.
- The path of the `streamflow.yml` file inside the container (e.g. `/streamflow/project/streamflow.yml`) must be passed as an argument to the Docker container.

A full example of a StreamFlow execution in a Docker container showing all these mounts is provided in Figure 35.

It is also possible to execute the StreamFlow container as a job in Kubernetes. In this case, StreamFlow is able to deploy Helm models directly on the parent cluster using `ServiceAccount` credentials. In order to do that, the `inCluster` option must be set to `true` for each involved model in the `streamflow.yml` file.

```
docker run -d \  
  --mount type=bind,source="$(pwd)"/my-project,target=/streamflow/project \  
  --mount type=bind,source="$(pwd)"/results,target=/streamflow/results \  
  --mount type=bind,source="$(pwd)"/tmp,target=/tmp/streamflow \  
  alphaunito/streamflow \  
  /streamflow/project/streamflow.yml
```

Figure 35 Streamflow docker execution (example).

## 5.4 An EDDL/ECVL Pipeline Example in StreamFlow

To demonstrate how the StreamFlow framework can be used to manage applications using the DeepHealth libraries in the ODH cloud-based platform, we present how to describe and run an example pipeline in the framework. The example selected is a use case pipeline<sup>26</sup> taken from project test cases. Here we are not focusing on the details of the specific ML algorithm implemented and the network used, but just on the integration mechanisms binding the DeepHealth libraries with the ODH platform and, most of all, how these bindings are defined and managed in the StreamFlow framework.

The use case pipeline we selected uses the EDDL and the ECVL to train a CNN on three different datasets (MNIST, ISIC and PNEUMOTHORAX), applying different image augmentations, for both the

<sup>23</sup> <https://github.com/alpha-unito/streamflow>

<sup>24</sup> <https://pypi.org/project/streamflow/>

<sup>25</sup> <https://hub.docker.com/r/alphaunito/streamflow>

<sup>26</sup> [https://github.com/deephealthproject/use\\_case\\_pipeline](https://github.com/deephealthproject/use_case_pipeline)

classification and the segmentation task. For our test case, which we call **odh-demonstrator**, we decided to perform a computation on the ISIC dataset, composed of two tasks:

- A SKIN\_LESION\_CLASSIFICATION\_TRAINING task, that trains the neural network loading the dataset in batches (needed when the dataset is too large to fit in memory);
- A SKIN\_LESION\_CLASSIFICATION\_INFERENCE task, that perform inference on classification task loading weights from the previous training process.

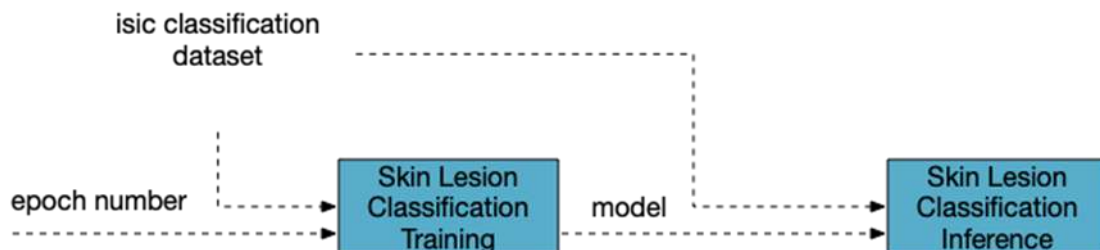


Figure 36 Odh-demonstrator workflow.

### 5.4.1 The Workflow

As described in the previous paragraph, StreamFlow needs the description of both the application workflow and the target infrastructure, the model. The application workflow is described using the cwl language in the `main.cwl` file (Figure 37). *Inputs* of the overall workflow are the number of epochs for the training process and the image dataset.

Looking at the `main.cwl`, workflow steps are:

- *training* that takes as input the number of epochs and the image dataset and produces a model.
- *inference* that takes as input the image dataset and the model produced in the previous training step.

Details for each step are, in turn described using cwl files, respectively `training.cwl` and `inference.cwl` (Figure 38). Looking at these files we can notice that the image dataset is specified not only in inputs but also as arguments for both tasks and it is specified referring to a description file (`isic_classification.yml`). The model, is instead defined as an `outputBinding` for the training task and as an `inputBinding` for the inference one. This way, the data dependencies between the two steps of the workflow and the external inputs are clearly defined.

Also, for both tasks the `baseCommand` (in this case a Python command) that has to be invoked to activate the task in the target container is defined, also providing additional configurations (in this case the option for selecting task execution on GPU).

```
#!/usr/bin/env cwl-runner
cwlVersion: v1.1
class: Workflow
$namespaces:
  sf: "https://streamflow.org/cwl#"

inputs:
  epochs: int
  dataset: Directory

outputs: {}

steps:
  training:
    run: training.cwl
    in:
      epochs: epochs
      dataset: dataset
    out: [model]
  inference:
    run: inference.cwl
    in:
      dataset: dataset
      model: training/model
    out: []
```

Figure 37 Main.cwl (odh-demonstrator).

<pre>cwlVersion: v1.1 class: CommandLineTool \$namespaces:   sf: "https://streamflow.org/cwl#"  baseCommand: ["python3", "/use_case_pipeline/ skin_lesion_classification_training.py", "--gpu"] arguments:   - position: 2     valueFrom: \$(inputs.dataset.path)/     isic_classification.yml  inputs:   epochs:     type: int     inputBinding:       prefix: --epochs       position: 1   dataset:     type: Directory  outputs:   model:     type: File     outputBinding:       glob: "isic_class_checkpoint.bin"</pre>	<pre>cwlVersion: v1.1 class: CommandLineTool \$namespaces:   sf: "https://streamflow.org/cwl#"  baseCommand: ["python3", "/use_case_pipeline/ skin_lesion_classification_inference.py", "--gpu"] arguments:   - position: 1     valueFrom: \$(inputs.dataset.path)/     isic_classification.yml  inputs:   dataset:     type: Directory   model:     type: File     inputBinding:       position: 2  outputs: {}</pre>
--	--

Figure 38 Training.cwl and Inference.cwl (odh-demonstrator).



To complete the workflow specification, the settings field is also defined referring to a config.yml file where the values of the input parameters are set (Figure 39).

```
epochs: 5
dataset:
  class: Directory
  location: /dataset/isic_classification
```

Figure 39: config.yml (odh-demonstrator)

## 5.4.2 The Model and Bindings

As already explained, the target model describes the multi-container environment where the workflow will be executed. In the case of the ODH platform the target is a Kubernetes environment, so StreamFlow will use a Helm Connector (type: helm). As a first step we define a helm-odh-demonstrator model, including a pylibs container with DeepHealth's Python libraries as service. In this case, the inCluster flag is set to true because we are running StreamFlow in Kubernetes too. Finally, in the bindings list the helm-odh-demonstrator is associated to both workflow steps, so that the same pylibs container will be used for the entire workflow.

The execution environment of the pipeline use case is depicted in Figure 40 where the blue layer represents the odh-demonstrator Kubernetes cluster. The StreamFlow manager is running in a Pod on a CPU node and orchestrates the execution of the two sequential pipeline tasks, deployed in turns in the pylibs containers and running on GPU nodes.

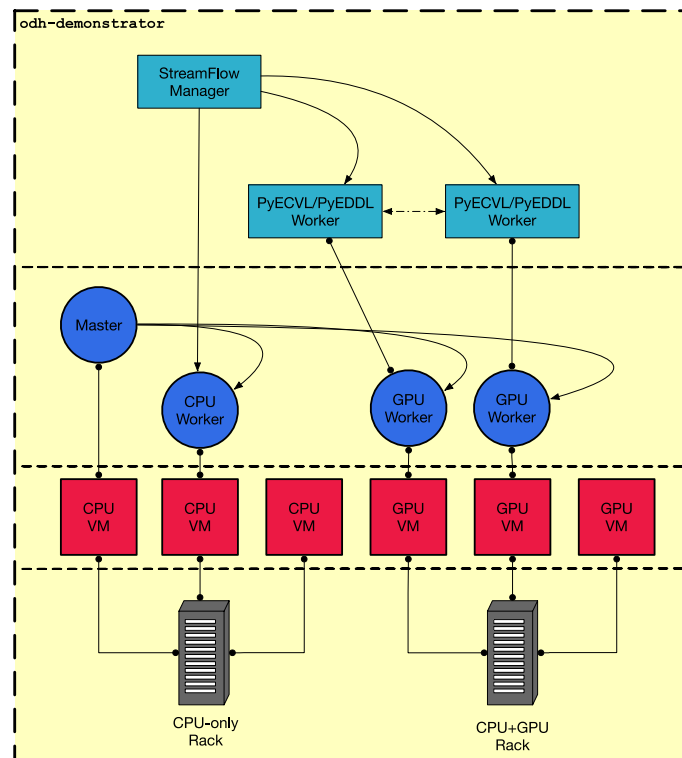


Figure 40 Odh-demonstrator execution environment.

It is worth noting that the same pipeline could be easily deployed also in a different target infrastructure, only providing the correct model description.

Also, it should be considered that this is just a simple demonstrator and further activity is ongoing to enhance integration between the DeepHealth toolkit and the ODH platform:

- At infrastructure level, mechanisms to guarantee persistent volume management inside the Kubernetes cluster are being investigated;
- Authentication/authorization mechanisms for executing application workflows on hybrid infrastructures with seamless access are considered;
- Support tools for enabling users to define StreamFlow description files should be defined, likely based on templates definition;
- StreamFlow integration with DeepHealth backend and frontend should be evaluated;
- Integration in ODH of the new releases of DeepHealth libraries, also comprising distributed and parallelization mechanisms is foreseen.

## 6 Overheads and Drawbacks of Deep Learning on the Cloud

In this section we will quantify the overhead introduced by the containerization of the DeepHealth libraries in realistic usage scenarios. Since the implementation of all the use cases is not yet complete, we will rely on the Use Case Pipeline repository<sup>27</sup> and the network architecture developed for UC13 (epileptic seizure detection) as test benches.

The tests were focused on the CPU version of the library, but UC13 was also evaluated on GPU. The system used for the benchmarks was a computing server with 2 Dual Intel Xeon Gold 6154 CPUs (36 cores, 72 threads), with a maximum processor speed of 3.70 GHz and 384 GB of RAM memory, while the GPU is a Nvidia Tesla T4 with 16 GB of memory. The installed operating system is CentOS v7.7.

### 6.1 Test Case 1: MNIST Pipeline

The first test case to evaluate the overhead of the container images of the libraries was taken from the Use Case Pipeline repository<sup>28</sup>, which can be considered the reference example for the practical integration of the EDDL and the ECVL. This repository provides several use cases, one of which is based on the popular MNIST dataset<sup>29</sup>. In this test case we used two network architectures: a classical LeNet structure<sup>30</sup> implemented in the EDDL library (v0.4.4), and an autoencoder architecture implemented in the PyEDDL library (v0.7.0).

Table 2 Overhead introduced by the Docker Images in training a model with the MNIST dataset shows the required training time per epoch using different numbers of CPU cores, for both containerized and regular execution scenarios. In this case, the dockerized version of the application does not experience any overhead, and indeed the running times are sometimes slightly shorter – which can likely be attributed to variability in the experimental apparatus. These measurements corroborate with the expected low overhead of containerizing these workloads.

Table 2 Overhead introduced by the Docker Images in training a model with the MNIST dataset

Library	Dockerized	Number of CPU Cores	Time per epoch (s)	Overhead	Nº Measurements
EDDL + ECVL	No	36	396 ± 6.67	Reference	100
EDDL + ECVL	Yes	36	379 ± 4.84	-4.3%	100
PyEDDL	No	4	29.61 ± 5.38	Reference	100
PyEDDL	Yes	4	32.36 ± 7.99	9.3%	100

<sup>27</sup> [https://github.com/deephealthproject/use\\_case\\_pipeline](https://github.com/deephealthproject/use_case_pipeline)

<sup>28</sup> [https://github.com/deephealthproject/use\\_case\\_pipeline](https://github.com/deephealthproject/use_case_pipeline)

<sup>29</sup> [https://en.wikipedia.org/wiki/MNIST\\_database](https://en.wikipedia.org/wiki/MNIST_database)

<sup>30</sup> <https://en.wikipedia.org/wiki/LeNet>

PyEDDLl	No	1	$7.47 \pm 0.02$	Reference	100
PyEDDLl	Yes	1	$7.45 \pm 0.02$	-0.3%	100

These numbers are certainly surprising, and have revealed an issue in the current multi-core implementation of the EDDLl. Specifically, the configuration parameter that sets the number of CPU cores to use is not well behaved and it causes unexpected performance drops. This issue has been acknowledged and will be addressed in the near future. Nevertheless, the problem does not affect the evaluation of the containerization overhead, which is the target parameter of these experiments.

## 6.2 Test case 2: Epileptic Seizure Detection Network

A second set of tests was performed training the neural network designed for UC13 (Epileptic seizure detection on EEG signals) using the public CHB-MIT database<sup>31</sup> as dataset. The network architecture, shown in Figure 41, is relatively simple: it consists of 5 convolutional layers followed by 2 dense layers. The input is just the raw multi-lead EEG signal, so no preprocessing is required and therefore only the EDDLl library is evaluated. Specifically, the implementation was done using the PyEDDLl interface. For the sake of comparison, we also include the performance measurements obtained from the evaluation of the same network architecture implemented using the Keras library. The results for the training task with the whole dataset are shown in Table 3.

In this specific use case, results show a comparable and significant overhead in the containerized version using 1 CPU core, for both Keras and PyEDDLl. However, while for Keras the overhead is reduced proportionally to the number of cores used, for PyEDDLl the overhead increases in multi-core settings – which is in direct contrast to the low overhead on a very high number of cores observed in Test Case 1. Both the observed overhead and the multi-core effects are unexpected and need to be clarified with further experimentation. The causes might be rooted in some particular aspect of this neural network architecture or perhaps even some particular (perhaps transient) conditions at test execution time. These experiments will be extended to the different use cases as they integrate the DeepHealth toolkit, allowing us to better characterize the efficiency of the containerized DeepHealth libraries and address efficiency problems that might be identified. Regarding the GPU implementation, observed containerization overhead is practically inexistent for PyEDDLl (0.2%) and relatively low for Keras (8.7%).

Compared with the Keras library, for UC13 the PyEDDLl is consistently between 20 and 30 times slower on CPU, except on the multi-core dockerized version, in which the running time is around 60 times slower. This difference is likely linked to some issue that is specific to this network and the maturity of the Keras implementation with respect to the EDDLl library, and we expect the gap to narrow as work progresses in DeepHealth. On the other hand, on GPU the difference in execution time are reduced to 8 times slower

<sup>31</sup> <https://physionet.org/content/chbmit/1.0.0/>

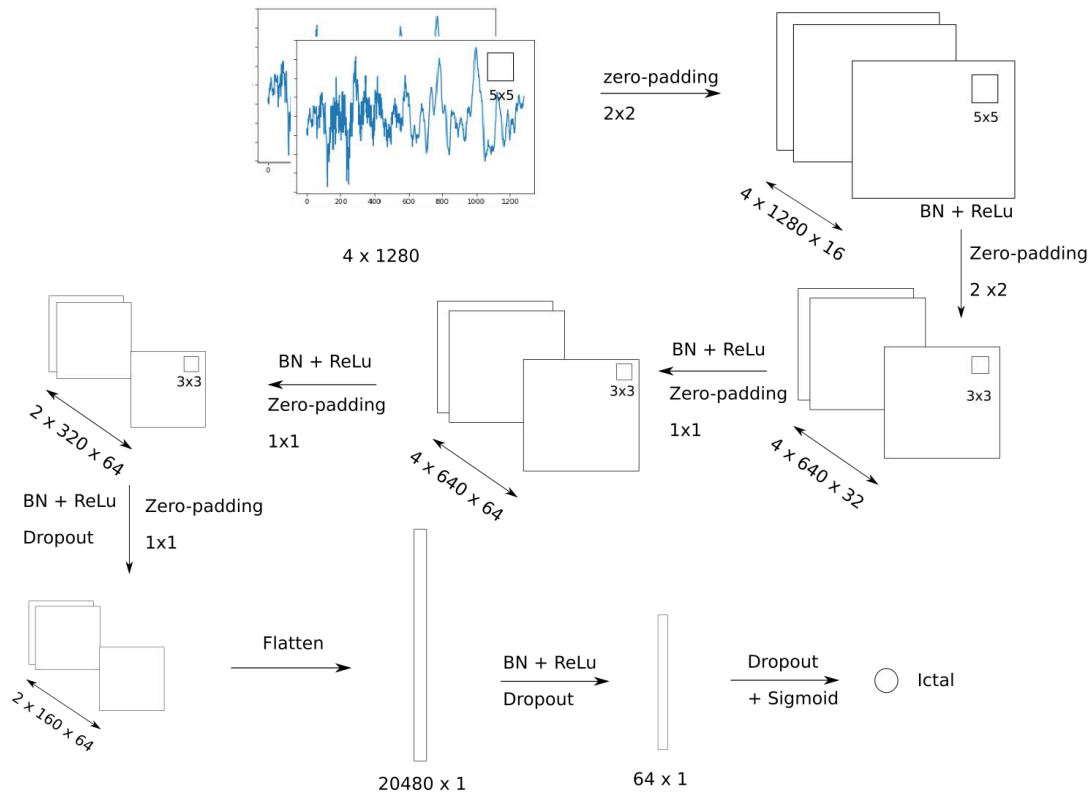


Figure 41 Network architecture for Epileptic Seizure Detection.

Table 3 Overhead introduced by the Docker images in the training of UC13.

Library	Dockerized	CPU/GPU	Time per epoch (s)	Overhead	Nº Measures
PyEDDLL	No	CPU 1 core	452 ± 7,09	Reference	150
PyEDDLL	Yes	CPU 1 core	853 ± 8,20	88,8%	50
PyEDDLL	No	CPU 4 cores	246 ± 4,66	Reference	150
PyEDDLL	Yes	CPU 4 cores	667 ± 7,50	170,7%	50
PyEDDLL	No	GPU	5,569 ± 0,02	Reference	750
PyEDDLL	Yes	GPU	5,580 ± 0,02	0,2%	750
Keras	No	CPU 1 core	17,09 ± 0,37	Reference	750
Keras	Yes	CPU 1 core	29,27 ± 0,75	71,3%	750
Keras	No	CPU 4 cores	11,05 ± 0,35	Reference	750
Keras	Yes	CPU 4 cores	13,03 ± 0,59	17,9%	750
Keras	No	GPU	0,724 ± 0,31	Reference	750
Keras	Yes	GPU	0,787 ± 0,23	8,7%	750

## 7 Conclusions

---

This deliverable reports the activities performed in Task 2.4, which had the objective to facilitate and demonstrate the use of the DeepHealth EDDLL library on cloud computing infrastructure. The activities gone beyond this objective to allow users to use both DeepHealth libraries together on the cloud for the creation of complete cloud-enabled deep learning pipelines.

A full spectrum of solutions has been delivered to run the DeepHealth toolkit on the Kubernetes platform. At the lower level, Docker container images have been provided. These form the basis for the results presented in this report, but also for the integration of DeepHealth components in the cloud-enabled DeepHealth platforms – in addition, of course, to external adopters. At a higher level, the DeepHealth front end has been ported to the cloud and the DeepHealth libraries have been integrated into the ODH platform and the StreamFlow workflow manager, offering ready-to-use cloud-enabled solutions for expert users. From a scalability perspective, the EDDLL has been extended to be able to efficiently exploit large-scale multi-cloud and hybrid cloud infrastructures, and this opportunity has been made available to consortium partners through the deployment of on-premise private cloud. Finally, continuous integration pipelines have been put in place to ensure that as the development of the DeepHealth libraries continues, those improvements will be automatically integrated into new container images so that the solutions described in this document remain up-to-date and sustainable in time.