DEEPHEALTH

D3.6 ECVL adaptation to cloud environments

Project ref. no.	**H2020-ICT-11-2018-2019 GA No. 825111**
Project title	Deep-Learning and HPC to Boost Biomedical Applications for Health
Duration of the project	1-01-2019 – 31-12-2021 (36 months)
WP/Task:	WP3/T3.3
Dissemination level:	PUBLIC
Document due Date:	31/05/2020 (M17)
Actual date of delivery	29/05/2020 (M17)
Leader of this deliverable	TREE
Author (s)	David Gonzalez (TREE), Luca Pireddu (CRS4), Tatiana Silva (TREE), Marco Enrico Piras (CRS4), Maria A. Serrano, Eduardo Quiñones (BSC), Barbara Cantalupo (UNITO), Marina Zapater (EPFL), Francesco Versaci (CRS4), Tomas Teijeiro (EPFL), Jose Ramón Hervas (TREE)
Version	V7.0

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 82511

Document history

Version	Date	Document history/approvals
1	14/04/2020	Outline proposal
2	29/04/2020	Collected first draft of contents from CRS4, TREE, UNITO, BSC and EPFL
3	11/05/2020	Complete draft version
4	15/05/2020	Internal review for CEA
5	18/05/2020	Complete version of the deliverable
6	26/05/2020	Review by technical manager (Jon Ander Gómez) and PC (Monica Caballero)
7	28/05/2020	Final version

Table of contents

1 Executive Summary

This report for D3.6 presents the results from the activities of T3.3, *ECVL Adaptation to Cloud Environments*. The goal of this task was to extend the ECVL 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 infrastuctures. 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 ECVL and the companion EDDL library (T2.4 - *EDDLL 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 ECVL 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 Open DeepHealth (ODH) platform and the StreamFlow workflow manager, offering ready-to-use cloud-enabled solutions for expert users. From a scalability perspective, distributed inference operations on cloud infrastructure have been implemented and demonstrated, and cloud resources have been made available to consortium partners through the deployment of on-premise private cloud. Also, a preliminary analysis of the overheads of containerization incurred by the ECVL have hinted they are next to null, meaning that the flexibility and scalability of the cloud-enabled solutions presented in this report do not incur a significant performance penalty in terms of processor cycles. 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 EDDLL 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 relavant 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 3.3, which aimed to facilitate and demonstrate the use of the DeepHealth ECVL 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.

The discussion around how to best support the use of DeepHealth, and the ECVL in particular, on cloud computing resources resulted in the decision to target the Kubernetes (k8s) container orchestrator as the DeepHealth 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[1]) 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 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[2], Kops[3]). 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 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 ECVL. Specifically, Section 2.2 summarizes the description of 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 ECVL and PyECVL DeepHealth toolkit components, as well as the continuous integration system that has been put in place to automatically generate new up-to-date images as the development of these components advances throughout the rest of the project. In Section 4 we describe how the DeepHealth toolkit, including the front-end application described in D2.5 and D3.5, have been extended to run on Kubernetes. Next, Section 5 describes the integration of ECVL into the Open DeepHealth (ODH) Platform. Finally, Section 6 provides an analysis of the efficiency costs payed 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 D2.6 - EDDLL Adaptation to Cloud Environments

As an introductory note, it is important to highlight the relation between this report and the complementary report *D2.6 EDDLL adaptation to cloud environments*. While the original DeepHealth work plan structures the EDDLL- and ECVL-related activities as distinct entities, advancement in the project has revealed advantageous to adopt a solution where the EDDLL and ECVL tightly interoperate within the DeepHealth toolkit. For instance, consider how any image-based DeepHealth model training or inference process performed by the EDDLL 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 EDDLL- and ECVL-specific tasks T2.4 and T3.3, which are respectively reported in D2.6 and in this D3.6, are in many ways analogous as they solve the extended problem of facilitating the use of both the EDDLL and the ECVL *together* on the cloud. In the interest of avoiding content duplication, when deemed appropriate these analogous results are described in detail in only one of the two reports, while the other presents a summary. On the other hand, results that focus on one of these components are naturally reported only in the one specific report.

2.2 The Kubernetes Platform

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 system complexity increases – e.g., complex multi-component software applications, multi-node clusters – running deployments become increasingly difficult. Kubernetes supports the automation of much of the work required to maintain and operate such complex services in a distributed environment. D2.6 includes a brief summary of the Kubernetes platform which may be useful to better understand its architecture.

As explained in Section 1.1, Kubernetes was chosen as the target cloud platform for the DeepHealth toolkit. An on-premise Kubernetes cluster has been built by TREE and made available to the DeepHealth consortium partners. It will be part of a hybrid cloud environment (developed within T5.6), using AWS services. To support resource management tasks, the open-source Rancher[1] tool was used, to support the management of the operational and security challenges on multiple interconnected Kubernetes clusters across any infrastructure. In addition, an API was developed to facilitate the deployment of processes and the management of workflows in hybrid cloud scenarios. This API effectively makes the source of the computing resources being used transparent to the user – be the resources on premise or on public clouds. More details about the API and the use of Rancher are available in *D2.6 EDDLL adaptation to cloud environments*.

3 Container Images and Continuous Integration

Since the DeepHealth cloud activities target the Kubernetes *container orchestration* platform, packaging the DeepHealth toolkit in effective container images is key for enabling its operation on the cloud. Container images 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, a continuous integration pipeline has been put in place to automatically keep them up-to-date with new releases of the DeepHealth software components. Because the DeepHealth components tightly interoperate to provide functionality for deep learning pipelines, we have implemented a solution that contemplates the toolkit in its entirety and we have adopted the same principles in designing a solution for the ECVL as we did for the EDDLL. Therefore, in the interest of avoiding content duplication, this section will provide a summary of the achievements in terms of container images and continuous integration in DeepHealth and, where appropriate, will focus on particular issues pertaining to the ECVL and PyECVL; on the other hand, for further details, especially pertaining to the rationale behind some design decisions, we refer the reader to this report's sister deliverable D2.6.

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. The resuling set of images and their interdependence is illustrated in Figure 1. As the figure shows, library-specific images are generated for both ECVL and PyECVL. Moreover, two compound images are created: the `libs` image containing the entire DeepHealth C++ runtime (EDDLL and ECVL) and the `pylibs` image packaging the entire Python runtime. The latter two compound images are the recommended option to integrate deep learning functionality into cloud-based applications.

[1] https://rancher.com/docs/rancher/v2.x/en/

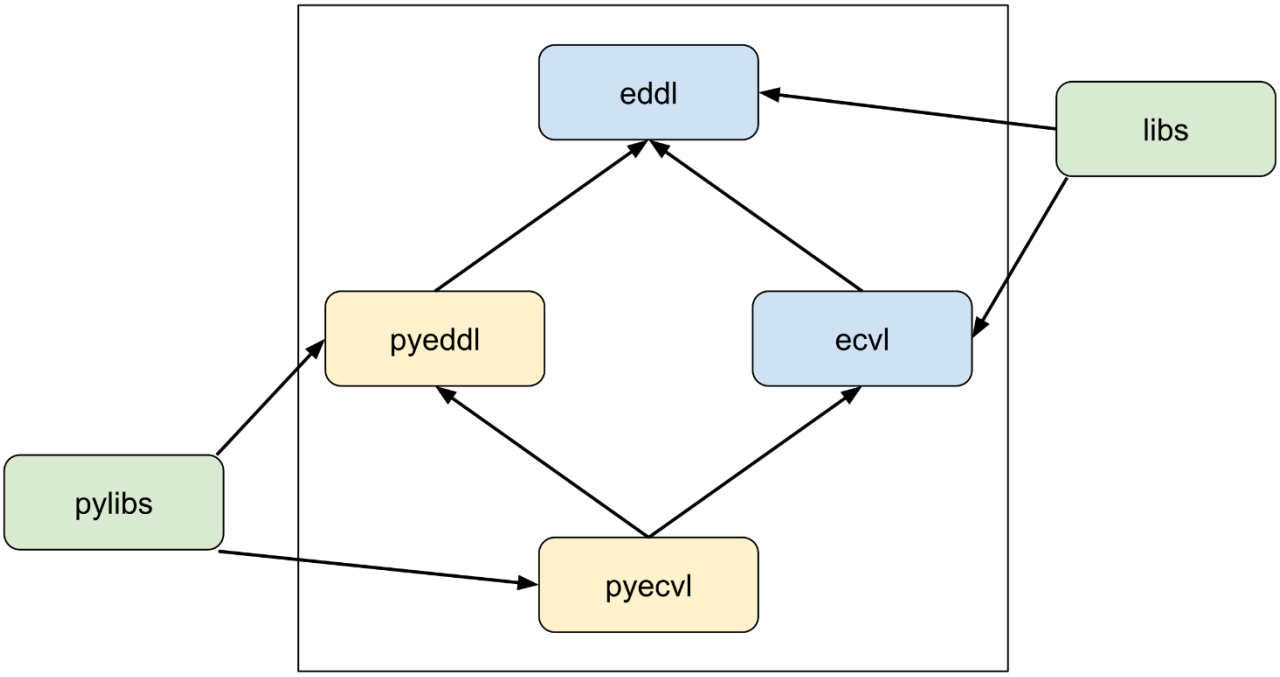

Figure 1 Relation between the main DeepHealth library Docker images. C++ API libraries are shown in blue, Python API in yellow. The green images conveniently package the pair of EDDL and ECVL libraries to support full pipelines.

The Docker images just described contain all the DeepHealth runtime requirements and are therefore ready to use to run DeepHealth-based applications that are grafted onto them. In addition, a set of toolkit Docker images has been created that adds the functionality required to build applications – i.e., library headers, compilers and other build tools, etc. – and are thus complete build environment for DeepHealth-based applications. For every DeepHealth runtime library image there is a companion toolkit image called library-toolkit (e.g., `pyeddl` and `pyeddl-toolkit`). As an example, Figure 2 illustrates how a program that uses the ECVL can be compiled without installing any supporting tool or libraries on the host computer by using the ecvl-toolkit image. Similarly, the toolkit images are useful for building Docker images of software that depend on DeepHealth components – for instance, through multi-stage Docker container builds.

```
$ docker run –it --rm -u $(id -u) -v $(pwd):/my_examples \
   dhealth/ecvl-toolkit:latest /bin/bash
$ cd /my_examples/
$ g++ example_imgproc.cpp -o example -std=c++17 -ldataset -lecvl_core –ldcmdata
-ldcmimage -ldcmimgle -ldcmjpeg -li2d -lijg8 -lijg12 -lijg16 -loflog –lofstd
-lopencv_core -lopencv_imgcodecs -lopencv_imgproc -lopencv_photo –lopenslide
-lecvl_eddl -leddl -lyaml-cpp -lstdc++fs -pthread
$ ./example
```

Figure 2 Example showing how the ECVL toolkit image can be used to compile and run ECVL client program residing on a host computer without locally installing any software other than Docker itself. Notice the extensive list of libraries required by the example; these are all packaged on the container image and thus their installation on the host is not required.

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. All 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 Image Versioning and Publication

The various Docker images described in the previous sections have different priorities. The library-specific images (`ecvl` and `pyecvl`) closely track the development of the individual DeepHealth components. For these, the DeepHealth continuous integration pipeline (described later in this document) automatically generates a new version of the image every time modifications are pushed to the corresponding library-specific software repository on GitHub. On the other hand, the compound images (`libs` and `pylibs`) provide snapshots of the libraries that have been tested to work well together. For them, a new version of the image is generated when the DeepHealth developers tag a release of the DeepHealth `docker-libs` repository[2], which contains the code implementing the Docker-related DeepHealth functionality.

Table 1 DeepHealth ECVL-related 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 triggered that is manually triggered.

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

The Docker images produced by the DeepHealth project are published on DockerHub[3]. A DeepHealth organization has been created[4] 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 other DeepHealth components such as the toolkit *front end*.

3.2 Continuous Integration Pipeline

In DeepHealth, automated pipelines have been implemented to support the development process with continuous integration (CI) of new versions of DeepHealth software into the Docker images, accompanied by the automated testing of those images. The full implementation is described in more detail in D2.6. The ECVL-related Docker images each have their own CI pipeline that runs on the Jenkins installation hosted by UNIMORE[5], next to the conventional DeepHealth CI pipelines. All DeepHealth CI pipelines are triggered by the contribution of new changes to the relevant code repositories on GitHub. Their implementations are based on the core DeepHealth Dockerized CI pipeline that has been created as part of this work and is published in the `deephealthproject/docker-libs` repository on GitHub. Specific adapters have been added to the core pipeline to handle the compilation and execution interface of the ECVL and PyECVL. The automatic compilation, testing and publication of these and all DeepHealth Docker images is integrated into the overall DeepHealth CI system and all components follow the same procedure comprised of three steps:

1. A new image is compiled following a change to the corresponding GitHub repository;

[2] https://github.com/deephealthproject/docker-libs
[3] https://hub.docker.com/
[4] https://hub.docker.com/orgs/dhealth
[5] https://jenkins-master-deephealth-unix01.ing.unimore.it

2. The tests for the component are executed within a Docker container, thus verifying that the containerized version of the software works;
3. If the tests pass, the image is published on DockerHub.

A thorough image tagging scheme ensure that the published Docker images can be unequivocally identified, for instance through the commit id from the code repository or from the specific release name that triggered the build.

Finally, the core CI pipeline code can be used by developers independently of any other components to easily build and test images locally – a particularly useful feature when developing new code for these components. The documentation of the `docker-libs` component includes a full list of the supported commands[6].

4 ECVL Support for Cloud Environments

4.1 Distributed Inference on the Cloud

The ECVL library has also been adapted to be used in a cloud environment by extending it to perform parallel and distributed jobs on Kubernetes-managed computing resources. In this section we demonstrate this functionality by using it to perform distributed inference on a dataset. It is important to note that Kubernetes itself is in charge of balancing the different JOBs within the cluster, distributing them on different machines depending on the resources available at execution time.

When working with the DeepHealth toolkit, datasets are represented using the DeepHealth Toolkit Dataset Format[7] (DTD). This is a simple and flexible YAML syntax to describe a dataset for consumption by the DeepHealth libraries (EDDLL/ECVL). In order to perform distributed inference, the idea is to split the dataset into several parts and execute each part separately. An implementation for doing this is available in the deephealthproject/deephealth-k8s repository on GitHub[8].

Thus, this engine receives the YAML representation of the dataset and divides it into n-parts (also written in DTD format). Once we have the multiple dataset files, each of them will be used as input for an inference operation that will be executed separately in the Kubernetes cluster. Running the inference with the same input parameters except for the n-part of the DTD file, it is possible to run (in parallel) the same job with different input files. In this way, the global operation can be completed much more quickly. Note that the parallelization of jobs is managed by Kubernetes, meaning it will schedule each Job/Pod to the node it considers best suited at execution time.

As an example of how this works, we can imagine having 2750 images to process. We can divide it into 5 parts, so we would have 5 subsets with 550 images each. If the dataset does not split evenly into the desired number of partitions, (e.g., 2750 in 4 splits = 687.5) the excess dataset items are placed in the last slice. An implementation done with a project dataset is exemplified in Section 4.1.1.

Currently, the user must choose the desired number of splits. We are working to implement a similar distribution strategy in COMPSs, so that the deployment is transparent to the programmer and we can target heterogeneous platforms.

4.1.1 Use Case Integration and Testing

We tested and demonstrated the efficacy of the dataset splitting technique by applying it to the DeepHealth Use Case 12 (Skin Cancer Melanoma detection) dataset. This dataset is composed by several skin lesion images; a more detailed description can be found in *D1.1 Use Cases requirements*. A pre-trained deep learning model was provided by the use case leaders in EDDLL binary format.

[6] https://github.com/deephealthproject/docker-libs#how-to-build-test-and-publish
[7] https://github.com/deephealthproject/ecvl/wiki/DeepHealth-Toolkit-Dataset-Format
[8] https://github.com/deephealthproject/deephealth-k8s/tree/master/split-yaml

The goal of the test was to perform the inference of a project use case dataset distributed over multiple pods/machines.

In the inference scenario explained in Section 4.1, the dataset is split into several parts and each one is executed in parallel in a different POD, either inside the same machine or distributed over multiple machines. A workflow script implementing the complete operation is available in the deephealthproject/deephealth-k8s GitHub repository[9].

Consider, for example, the requirement to run three parallel inference jobs. We thus need to divide the Dataset into three parts, the engine builds three YAML files, as described in the GitHub repository. The output log of this split engine would be as shown in Figure 3.

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
[DeepHealth] Start split-yaml process '/deephealth/dataset/isic_segmentation/isic_segmentation.yml'
[DeepHealth] Write '/deephealth/dataset/isic_segmentation/part-000-isic_segmentation.yml' file
[DeepHealth] Write '/deephealth/dataset/isic_segmentation/part-001-isic_segmentation.yml' file
[DeepHealth] Write '/deephealth/dataset/isic_segmentation/part-002-isic_segmentation.yml' file
[DeepHealth] Finish split-yaml process. Check the directory '/deephealth/dataset/isic_segmentation'
```

Figure 3 Split Dataset Engineer Log.

When the split engine is finished, and we have the three parts, one JOB will be launched for each part (Figure 4), with their respective PODs (Figure 5).

```
NAME                    COMPLETIONS   DURATION   AGE
dh-job-inference-000    1/1           12m        19h
dh-job-inference-001    1/1           12m        19h
dh-job-inference-002    1/1           15m        19h
dh-job-splityaml        1/1           34s        19h
```

Figure 4 Get JOBs.

```
NAME                          READY   STATUS      RESTARTS   AGE
dh-job-inference-000-zkdb9    0/1     Completed   0          19h
dh-job-inference-001-kz47l    0/1     Completed   0          19h
dh-job-inference-002-smh64    0/1     Completed   0          19h
dh-job-splityaml-cxjnz        0/1     Completed   0          19h
```

Figure 5 Get PODs.

As seen in Figure 4 and Figure 5, a JOB/POD of the split Dataset engine and one for each partition has been generated. However, even if the JOB is not completed and the POD is not finished, the log file will be available. The application log can be consulted as shown in Figure 6 and Figure 7.

[9] https://github.com/deephealthproject/deephealth-k8s/tree/master/use_case_pipeline_distriburted_inference

```
mkdir: cannot create directory '/deephealth/outputs/trash': File exists
Reading dataset
Testing
Batch 1/100 - IoU: 5.18135e-09 - IoU: 1.81127e-10
Batch 2/100 - IoU: 8.15661e-11 - IoU: 1.31303e-10
Batch 3/100 - IoU: 3.99521e-10 - IoU: 7.71545e-11
Batch 4/100 - IoU: 8.26173e-11 - IoU: 3.07598e-10
Batch 5/100 - IoU: 1.53374e-09 - IoU: 1.17028e-10
Batch 6/100 - IoU: 4.40141e-10 - IoU: 2.51256e-09
Batch 7/100 - IoU: 6.94444e-11 - IoU: 2.51319e-10
Batch 8/100 - IoU: 3.94166e-10 - IoU: 3.77316e-11
Batch 9/100 - IoU: 7.86596e-11 - IoU: 2.14731e-10
Batch 10/100 - IoU: 9.16086e-11 - IoU: 2.60417e-09
Batch 11/100 - IoU: 3.81679e-10 - IoU: 9.54745e-11
Batch 12/100 - IoU: 7.11744e-11 - IoU: 1.02145e-09
```

Figure 6 ECVL library log – part I

```
Batch 92/100 - IoU: 3.99042e-10 - IoU: 1.0523e-10
Batch 93/100 - IoU: 8.4317e-10 - IoU: 1.8044e-10
Batch 94/100 - IoU: 4.02091e-10 - IoU: 1.53704e-10
Batch 95/100 - IoU: 2.39234e-09 - IoU: 7.34646e-11
Batch 96/100 - IoU: 4.24755e-11 - IoU: 3.07503e-10
Batch 97/100 - IoU: 2.77393e-10 - IoU: 6.73854e-10
Batch 98/100 - IoU: 7.26427e-11 - IoU: 1.46843e-09
Batch 99/100 - IoU: 1.31234e-09 - IoU: 1.33529e-10
Batch 100/100 - IoU: 1.55497e-10 - IoU: 1.8423e-10
MIoU: 4.50062e-10
Generating Random Table
-----------------------------------------
input1          |  (3, 192, 192)=>      (3, 192, 192)
conv1           |  (3, 192, 192)=>      (64, 192, 192)
activation1     |  (64, 192, 192)=>      (64, 192, 192)
conv2           |  (64, 192, 192)=>      (64, 192, 192)
activation2     |  (64, 192, 192)=>      (64, 192, 192)
pool1           |  (64, 192, 192)=>      (64, 96, 96)
conv3           |  (64, 96, 96)=>      (128, 96, 96)
activation3     |  (128, 96, 96)=>      (128, 96, 96)
```

Figure 7 ECVL library log – part II

In the outputs folder, the outcome produced by the library is available (Figure 8).

```
[k8sadmin@deephealth1 outputs]$ ls -U
000  trash  001  002
[k8sadmin@deephealth1 outputs]$ ls -U 000 | head -10
batch_48_0_gt.png
batch_45_1_gt.png
batch_90_1_gt.png
batch_71_0_output.png
batch_64_1_output.png
batch_1_0_gt.png
batch_79_0_output.png
batch_20_0_gt.png
batch_86_0_output.png
batch_27_0_gt.png
[k8sadmin@deephealth1 outputs]$
```

Figure 8 Outputs.

In Figure 8 we see three folders (000, 001 and 002, one by each JOB) which contain the ECVL library output and a sample of ten elements of 000's folder.

The idea of parallelizing the inference work arises because this process was done sequentially using a pre-trained model. Since each answer of the inference is independent of the others, this possibility

is raised. In this way we increase performance and reduce time and costs. This relatively simple work parallelization and distribution strategy leverages the features built into Kubernetes to provide robustness. In fact, thanks to the fact that k8s automatically retries failed pods the system avoids increasing the probability of job failures with increasing numbers of splits. In addition, proper resource request configuration ensures that the k8s scheduler distributes work well across the available nodes. The existence of this use case leads us to extend it to future use cases that may arise, following this methodology of dividing the Dataset Format and building a workflow with this set of inputs.

4.2 Deployment of DeepHealth Front-End on Kubernetes

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. Together, these components are the DeepHealth front end, which allows expert users to exploit the DeepHealth library functionality without writing any programming code or locally installing the toolkit. These components are described in D2.5 *EDDLL Toolkit front-end* and D3.5 *ECVL Toolkit front-end*. As part of the activities to adapt the DeepHealth toolkit to the cloud, we have transformed the application into a distributed microservice architecture and ported it to the Kubernetes platform. The results of this work are described in this section.

4.2.1 Architecture of the DeepHealth Front-End

The DeepHealth front-end is a client-server system. The client GUI is a web-based application implemented with Angular[10] and runs in the user's web browser. Naturally, the client and the server do not need to be within the same infrastructure as the GUI (e.g., the server may be running in an institutional data center while the user connects from home through the internet). The server is actually a multi-component back-end system. The client application is delivered to the browser by a web server installed on the back-end. The client is configured to send requests to the RESTful API provided by the DeepHealth web service to implement its functionality. The web service delegates long-running operations, including DeepHealth-based ones (e.g., model training, inference), to a separate worker process implemented with Celery[11]. The web service and worker processes communicate through a RabbitMQ message broker and they also share a database and a file system where the data and logs reside.

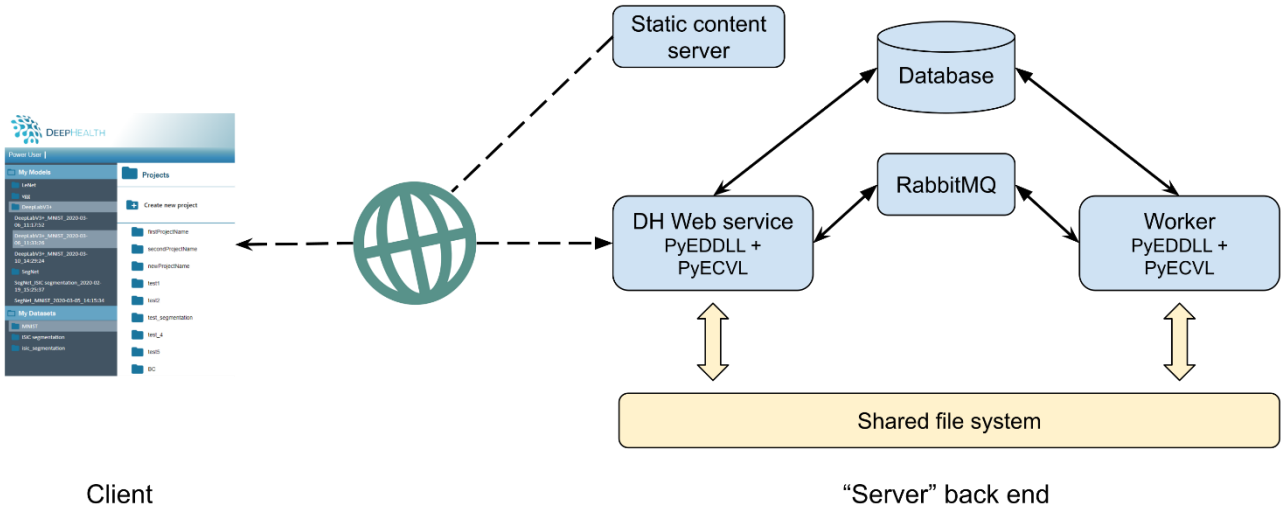

Figure 9 Architecture of the DeepHealth front-end system.

[10] https://angular.io
[11] http://www.celeryproject.org

4.2.2 Containerization of Components

Docker images have either been created from scratch or adopted for all the components of the front-end system. The implementation of the custom images is available in the `deephealthproject/docker-backend` repository on GitHub[12]. The images are published on DockerHub under the DeepHealth organization[13].

Static content server (`dhealth/frontend`). A Nginx server is used to serve the client application. A multi-stage image build has been implemented to avoid including the Node build environment in the runtime container image. Further, the client application was modified by CRS4 to enable the deployment-time configuration of the web service endpoint used by the GUI client. In fact, the original client application required this address to be set at compile time. Given the dynamic nature of cloud deployments, in most cases it is impossible to have this information before deployment. This change allows a single static Docker image containing the pre-compiled client to be used by all deployments.

DeepHealth web service and worker (`dhealth/backend`)**.** A new Docker image was created for both the DeepHealth web service and the worker microservice by extending the dhealth/pylibs image. The two components share a common code base, so it was natural to create a single dual-purpose image for them. A central container entrypoint program starts the appropriate role depending on the command that is invoked. The containerization of these components required adapting the way configuration settings are communicated to the application in a Kubernetes deployment (i.e., through Kubernetes `ConfigMaps`, `Secrets` and environment variables).

PostgreSQL Database, RabbitMQ. The PostgreSQL RDBMS and the RabbitMQ message broker are used by the DeepHealth back-end. These open source components are widely used and suitable community-supported Docker container images were readily available. Specifically, an image produced by Bitnami was used for PostgreSQL14 and the official image was used for RabbitMQ15. These images are well maintained and supported and are assured good future sustainability.

4.2.3 Prototype: docker-compose

A prototype of the Kubernetes deployment was created using docker-compose. Docker-compose provides non-distributed container orchestration that is much less sophisticated than Kubernetes, but still provides a useful platform for development and simple deployments. The prototype was used to accelerate the development and testing of the containerized components in the back end system and is available in the `deephealthproject/docker-backend` repository on GitHub.

4.2.4 Kubernetes Deployment

A Kubernetes deployment of the back-end was designed and implemented. The deployment, shown schematically in Figure 10 Schematic illustration of the Kubernetes deployment of the front-end system., maps the original static deployment of the system – i.e., the same macro components – to Kubernetes resource controllers. In addition, Kubernetes network access, storage, and role-based access control (RBAC) abstractions had to be implemented.

[12] https://github.com/deephealthproject/docker-backend
[13] https://hub.docker.com/u/dhealth
[14] https://hub.docker.com/r/bitnami/postgresql/
[15] https://hub.docker.com/_/rabbitmq

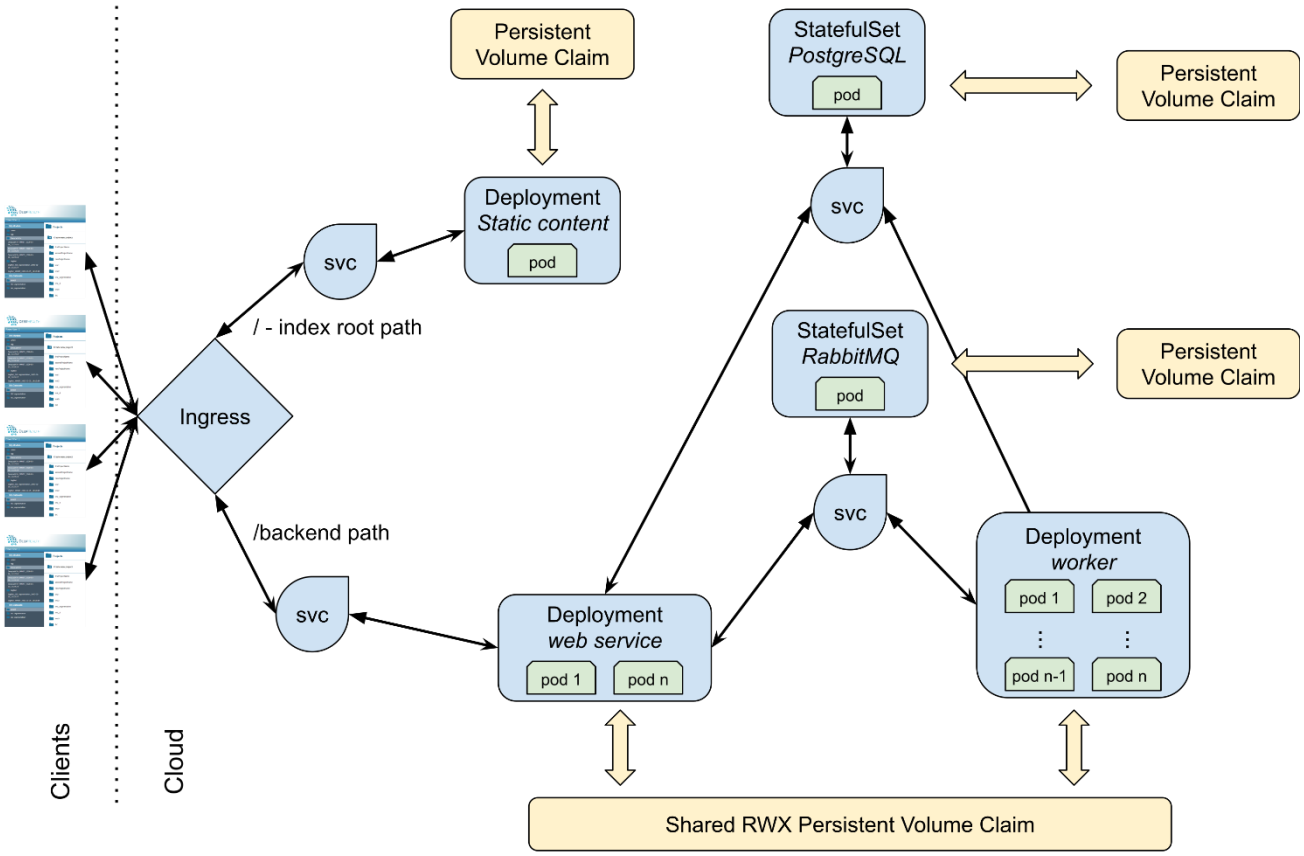

Figure 10 Schematic illustration of the Kubernetes deployment of the front-end system.

Microservices

The purpose-built containerized application microservices – i.e., web service, worker, static content server – are defined as Kubernetes *Deployment* resources. A Kubernetes deployment maintains a configurable number of identical pods, each running an instance of the microservice container. For those microservices that need to be accessed from other components (because they implement an API and/or serve content), a *Service* abstraction is defined that hides the multitude of identical pods behind a common service endpoint. This design allows the new k8s-based back-end to, for instance, multiply the number of concurrent web server processes to increase its processing capacity transparently to the client software.

On the other hand, given their stateful nature the PostgreSQL database and the RabbitMQ message broker are defined as *StatefulSet* resources. High-quality open source Kubernetes deployments for these latter two applications were already available from the community; these were used without modification.

Network Access

The Services that abstract access to the pods running our application containers are not automatically exposed to outside the Kubernetes cluster in a usable fashion. For this, we defined an appropriate Kubernetes *Ingress*, which allows access to the application on a configurable port. Moreover, it routes the requests to the appropriate service – be it to access the static content server or the DeepHealth back-end web service.

Storage

The original back-end application relied on the hosting server's local file system for all data. This storage strategy is not suitable for cloud deployments. Instead, in the new k8s deployment each

component is given its own *PersistentVolumeClaim*, except the web service and worker components which are given a single claim to share – since they need to see the same file system. Based on the specific Kubernetes cluster configuration, these claims will be mapped to the available storage systems (e.g., Cinder volumes on OpenStack, Elastic Block Storage on Amazon Web Services, etc.). The shared file system is different from the others in that it requests a volume that supports the read-write-many access mode, meaning that multiple pods can simultaneously mount the volume in read-write mode.

Jobs and Role-Based Access Control

The initialization and termination of the deployment is automated through the execution of Kubernetes *Jobs*. The initialization Job creates a single pod where two initialization steps are performed: the first is responsible for initialising the database, while the second collects all the back-end static files to be served by the static content server. As part of the database initialisation, any updates to the database schema are performed – implemented as Django migrations by the back-end software – thus simplifying the deployment of software updates to long-running DeepHealth system deployments. On the other hand, the termination Job is responsible for optionally releasing the k8s resources (e.g., PVCs, secrets) when the back-end deployment is deleted. Some of the tasks performed the initialization and termination jobs need to access the k8s API to manage the resources. Kubernetes controls access to these programming interfaces through Role-based Access Control (RBAC) policies, thus an ad-hoc RBAC resource has been created to allow the management jobs to handle the back-end PVCs and secrets.

4.2.5 Scaling

The DeepHealth front-end enabled the possibility of offering deep learning functionality as a service, thus hiding the complexity of the infrastructure behind a deployment away from the expert user and allowing them to focus solely on the deep learning problem at hand. However, the original static front-end application is not suitable for this type of use because it is limited to running a single job at a time, thus severely limiting its capacity to serve multiple users. On the other hand, Kubernetes deployment described here removes this limitation by intrinsically supporting scaling the number of instances of the various components of the application and running them on any nodes of the k8s cluster – thus distributing work across the available computing resources. For instance, the number of workers simultaneously running and ready to process user jobs can be ramped up to 25 with a simple command:

```
kubectl scale deployment/deephealth-backend-backend --replicas=25
```

We tested the efficacy of scaling the cloud deployment of the DeepHealth front end on CRS4's private OpenStack cloud. A Kubernetes cluster with 10 worker nodes was deployed, each with 7 virtual cores and 170 GB RAM. Then, a Ceph File System[16] was installed on the same cluster, using the computing nodes' local disks to create a POSIX-compliant shared file system that is compatible with the requirements of the DeepHealth back-end system. Finally, the DeepHealth back-end web service was installed on the cluster. Care was taken to ensure that the scheduler uniformly distributed worker over the nodes by configuring a `podAntiAffinity` criterion; also, resource requests were configured at 1 CPU per worker.

```
celery:
 affinity:
   podAntiAffinity:
     preferredDuringSchedulingIgnoredDuringExecution:
     - weight: 100
       podAffinityTerm:
         topologyKey: "kubernetes.io/hostname"
         labelSelector:
```

[16] https://docs.ceph.com/docs/master/cephfs/

```
            matchExpressions:
              - key: app.kubernetes.io/name
                operator: In
                values:
                  - deephealth-backend-celery
```

Figure 11 A podAntiAffinity scheduler criterion was used to configure the k8s scheduler to maximally distribute worker pods over nodes of the cluster, thus reducing unnecessary competition for resources.

We studied the weak scaling of inference of this setup on the standard MNIST dataset (which contains 10000 28x28 test images of handwritten digits) using a LeNet5 convolutional neural network. In our tests we measured the global completion time (max) of *n* identical, concurrent jobs for *n*=10, 20, 30 and 40 – i.e., corresponding to a load of 1, 2, 3, and 4 concurrent jobs on each single node. We expect the global completion time to increase as a function of the load, since all the nodes access the same file system and jobs on each (e.g., last-level cache (LLC), network). The results Figure 12 confirm this hypothesis, showing a linear increase of the completion time up to 44% moving from load 1 to 4 while keeping a very low variance (<2%) in the execution times among different runs. This amounts to about 15% overhead every time we increment the load.

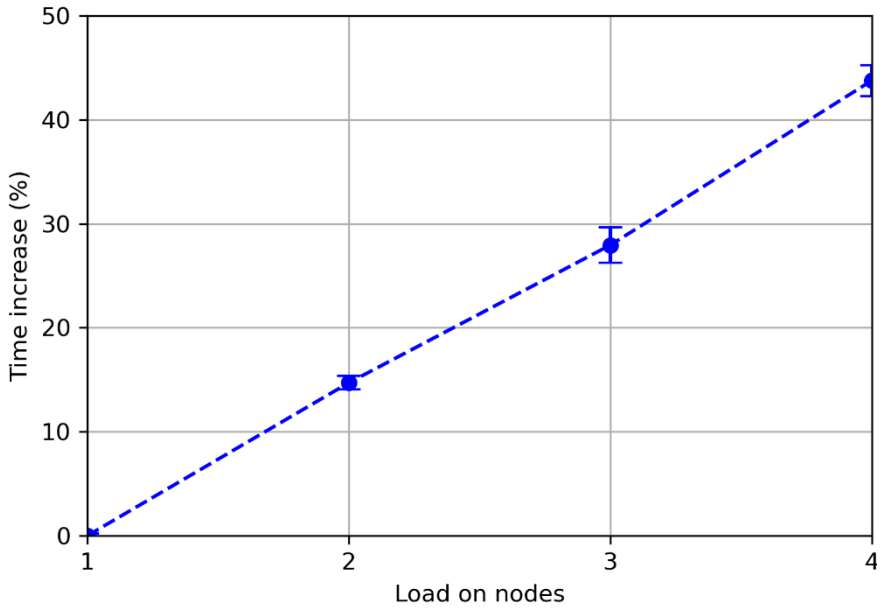

Figure 12 Running multiple concurrent inferencing jobs on the cloud-deployed DeepHealth back end. The plot shows a slight linear increase in global completion time with the number of jobs running on the same node.

4.2.6 Helm Charts

Kubernetes application deployments can be quite complex, and certain aspects must be customized for the specifics of the supporting Kubernetes cluster or cloud infrastructure, in addition to the particular user requirements (e.g., how much storage to use?). The Helm tool[17] provides package management functionality for the Kubernetes platform, facilitating reuse and parametrization of application deployments. Helm packages are called *charts*.

We have packaged the Kubernetes application deployment described in this section into two Helm charts:

- `deephealth-backend`: installs the DeepHealth web service and all the components it requires to operate;

[17] https://helm.sh

- `deephealth-frontend`: installs the `DeepHealth-backend` chart plus the static content server that serves the client application with the GUI.

These charts parameterize all relevant settings, allowing them to be used in any Kubernetes context. The full gamut of available parameters is described in the documentation available directly in the GitHub repository[18].

To publish these charts, we have created a DeepHealth Helm chart repository (accessible here by the Helm tool[19], add "/index.yaml" to the address see the contents in the browser). Thus, the installation of the DeepHealth front end can be a simple three-command process:

```
helm repo add dhealth https://deephealthproject.github.io/helm-charts/
helm repo update
helm install dhealth/deephealth-backend
```

Naturally, for a real deployment the user will need to tailor settings to the specific Kubernetes cluster (such as configuring the storage to use, domain name for the ingress, computing resources to allocate, etc.).

5 Deployment-as-a-Service on ODH Platform

As it has been introduced in Section 2.1, the EDDLL and ECVL tightly interoperate within the DeepHealth toolkit, and the technological solutions adopted for their integration are quite similar. In the context of the OpenDeepHealth (ODH) platform, this is particularly true, because one of the main goals of the overall framework is to ensure portability of ML applications across different environments therefore avoiding requiring specific adaptation for the target infrastructure. From the platform perspective, both EDDLL and ECVL are managed in the same way, based on a framework relying on multi-container technology. For this reason, in this section, we are only providing an overview of the mechanisms developed to provide Deployment-as-a-Service capability on ODH platform, referring the reader to the deliverable D2.6, and others if needed, for an in-depth description of the technology adopted and developed.

The OpenDeepHealth (ODH) platform, designed to implement the DeepHealth project requirements, has been developed as part of the University of Turin's HPC4AI[20] infrastructure, which is composed of a federated OpenStack[21] cloud with multi-tenant private Kubernetes instances. The ODH platform (see Figure 13) is defined as an HPC secure tenant where a multi-tenant Kubernetes Container 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*.

Besides the Kubernetes cluster, at a higher level of abstraction, UNITO is developing StreamFlow[22], a novel workflow model that has been developed for managing the deployment and the execution of tasks in multi-container environments. StreamFlow allows exploiting container's portability properties to simplify the execution of distributed applications based on DeepHealth libraries, on different and possibly hybrid infrastructures. To this end, a list of Docker containers, including DeepHealth toolkit and libraries, is deployed in the ODH platform (see Figure 13) supporting ML application development and execution. Referring to StreamFlow models allows deploying and executing any application that

[18] https://github.com/deephealthproject/docker-backend#parameters
[19] https://deephealthproject.github.io/helm-charts
[20] 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
[21] 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.
[22] StreamFlow: cross-breeding cloud with HPC Published in ArXiv 2020 https://arxiv.org/abs/2002.01558

is integrated with them, and therefore, any application or tasks that is integrated into the DeepHealth library containers.

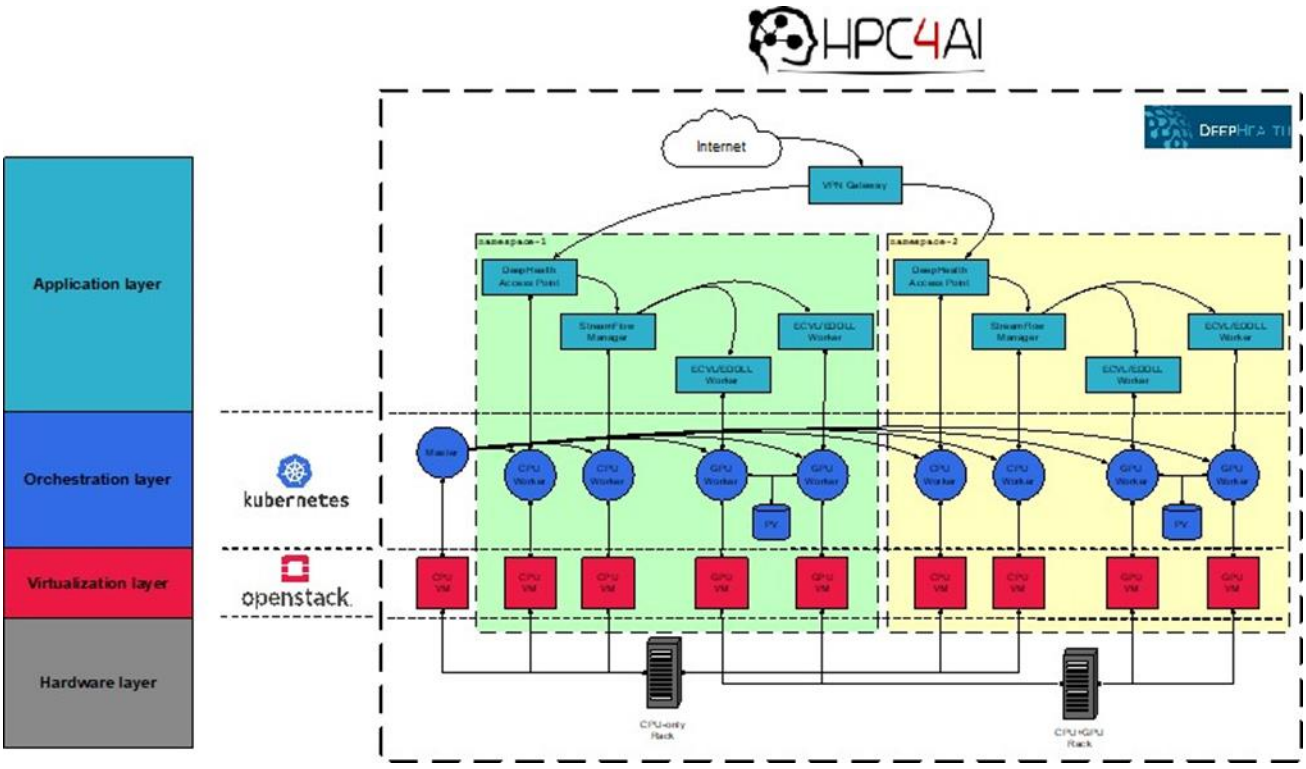

Figure 13: OpenDeepHealth platform

To summarize, the adaptation of ECVL, as well as EDDL, to ODH platform is based on the following pillars:

- Platform based on Kubernetes multi-container environment;
- DeepHealth toolkit distribution through Docker container technology;
- Application deployment and execution through StreamFlow workflow management.

A detailed description of the StreamFlow framework, its use with the DeepHealth toolkit and a use case example are provided in D2.6.

6 Overheads and Drawbacks of Computer Vision on the Cloud

Since at the time of writing the integration of the different use cases with the ECVL is in a very early stage, we will follow the same strategy explained in Section 6 in D2.6, and assess the overhead of the containerization of the ECVL on the applications provided by the Use Case Pipeline repository[23]. The hardware settings for the experiments were the same as described in D2.6.

The benchmarked applications were MNIST digit classification[24] and skin lesion classification and segmentation using the ISIC datasets[25] used in UC12 Table 2.

[23] https://github.com/deephealthproject/use_case_pipeline
[24] https://en.wikipedia.org/wiki/MNIST_database
[25] https://www.isic-archive.com/#!/topWithHeader/tightContentTop/challenges

Table 2 Overhead measured in the Docker versions of applications using the ECVL.

Application	Dockerized	Number of CPU Cores	Time per epoch (s)	Overhead	Experiment
MNIST classification training.	No	1	396 ± 6.67	Reference	20 training, 5 epochs
MNIST classification training.	Yes	1	379 ± 4.84	-4.3%	20 training, 5 epochs
Skin lesion classification inference.	No	1	11156 ± 29.38	Reference	1 inference, 5 epochs
Skin lesion classification inference.	Yes	1	11067 ± 21.51	-1.0%	1 inference, 5 epochs
Skin lesion segmentation inference.	No	1	3290 ± 13.94	Reference	1 inference, 10 epochs
Skin lesion segmentation inference.	Yes	1	3276 ± 15.55	-0.45%	1 inference, 10 epochs

In general, we observe no performance degradations after the containerization of the full pipelines. In any case, the experiments need to be extended to the different use cases as they are integrated with the ECVL. This will allow a much more precise quantification of the observed overheads.

7 Conclusions

This deliverable reports on the results of the activities performed in Task 3.3, whose objective was to facilitate and demonstrate the use of the DeepHealth ECVL library on cloud computing infrastructure. The results of the activities have gone beyond this objective to enable the use not just of the ECVL, but 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, distributed inference operations on cloud infrastructure have been demonstrated, and cloud resources have been made available to consortium partners through the deployment of on-premise private cloud. Also, a preliminary analysis of the overheads of containerization incurred by the ECVL have hinted they are next to null, meaning that the flexibility and scalability of the cloud-enabled solutions presented in this report do not incur a significant performance penalty in terms of processor cycles. 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 relevant and up-to-date in time.