European Commission | Horizon 2020
European Union funding
for Research & Innovation

Big Data technologies and extreme-scale analytics

MARVEL

Multimodal Extreme Scale Data Analytics for Smart Cities Environments

D3.2: Efficient deployment of AI-optimised ML/DL models – initial version[†]

Abstract: The purpose of this deliverable is to describe the MARVEL Edge-to-Fog-Cloud framework as the deployment layer for the AI/DL MARVEL components. This framework incorporates the deployment logic that is hidden behind MARVdash, the proposed Kubernetes dashboard for instantiating services as orchestrated containers, and deploying them to desired execution sites based on optimisation strategy. The main goal of the optimisation strategy is for MARVEL components to be deployed into Kubernetes nodes based on their resource requirements and the resource offerings of the actual nodes. Moreover, this deliverable will describe methods for compressing machine learning algorithms/models based on the resources available at the edge (e.g., reducing the size and operation time of million-parameter deep learning models). Such compression could minimise the computational overhead on the edge servers.

Contractual Date of Delivery	30/06/2022
Actual Date of Delivery	30/06/2022
Deliverable Security Class	Public
Editor	*Manos Papoutsakis, Anthi Barmpaki,* *Manolis Michalodimitrakis (FORTH)*
Contributors	FBK, AUD, AU, TAU, CNR, UNS, ITML, INTRA, ATOS, ZELUS, STS
Quality Assurance	*Alessio Brutti (FBK)* *Goran Martic (UNS)*

[†] *The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957337.*

The *MARVEL* Consortium

Part. No.	Participant organisation name	Participant Short Name	Role	Country
1	FOUNDATION FOR RESEARCH AND TECHNOLOGY HELLAS	FORTH	Coordinator	EL
2	INFINEON TECHNOLOGIES AG	IFAG	Principal Contractor	DE
3	AARHUS UNIVERSITET	AU	Principal Contractor	DK
4	ATOS SPAIN SA	ATOS	Principal Contractor	ES
5	CONSIGLIO NAZIONALE DELLE RICERCHE	CNR	Principal Contractor	IT
6	INTRASOFT INTERNATIONAL S.A.	INTRA	Principal Contractor	LU
7	FONDAZIONE BRUNO KESSLER	FBK	Principal Contractor	IT
8	AUDEERING GMBH	AUD	Principal Contractor	DE
9	TAMPERE UNIVERSITY	TAU	Principal Contractor	FI
10	PRIVANOVA SAS	PN	Principal Contractor	FR
11	SPHYNX TECHNOLOGY SOLUTIONS AG	STS	Principal Contractor	CH
12	COMUNE DI TRENTO	MT	Principal Contractor	IT
13	UNIVERZITET U NOVOM SADU FAKULTET TEHNICKIH NAUKA	UNS	Principal Contractor	RS
14	INFORMATION TECHNOLOGY FOR MARKET LEADERSHIP	ITML	Principal Contractor	EL
15	GREENROADS LIMITED	GRN	Principal Contractor	MT
16	ZELUS IKE	ZELUS	Principal Contractor	EL
17	INSTYTUT CHEMII BIOORGANICZNEJ POLSKIEJ AKADEMII NAUK	PSNC	Principal Contractor	PL

Document Revisions & Quality Assurance

Internal Reviewers

1. *Alessio Brutti, (FBK)*
2. *Goran Martic, (UNS)*

Revisions

Version	Date	By	Overview
0.8	30/6/2022	Editors	Addressed comments from the PC.
0.7	21/6/2022	Editors	Addressed comments from IR1 and IR2.
0.6	20/6/2022	IR2 (FBK, UNS)	Comments from IR1 and IR2
0.5	17/6/2022	Editors	Addressed comments from IR1 and IR2.
0.4	10/06/2022	IR (FBK, UNS)	Comments from IR1 and IR2
0.3	06/06/2022	Editors	Consolidated version for internal review
0.1	19/04/2022	Editors	ToC.

Disclaimer

The work described in this document has been conducted within the MARVEL project. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957337. This document does not reflect the opinion of the European Union, and the European Union is not responsible for any use that might be made of the information contained therein.

This document contains information that is proprietary to the MARVEL Consortium partners. Neither this document nor the information contained herein shall be used, duplicated or communicated by any means to any third party, in whole or in parts, except with prior written consent of the MARVEL Consortium.

Table of Contents

List of Tables

List of Listings

List of Figures

List of Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
AT	Audio Tagging
AVAD	Audio-Visual Anomaly Detection
AVCC	Audio-Visual Crowd Counting
CCTV	Closed-Circuit Television
CLI	Command Line Interface
CPU	Central Processing Unit
DFB	Data Fusion Bus
DL	Deep Learning
DMT	Decision-Making Toolkit
DNN	Deep Neural Network
E2F2C	Edge to Fog to Cloud
GA	Grant Agreement
GNU	GNU's Not Unix
GPU	Graphics Processing/Processor Unit
HDD	Hierarchical Data Distribution
HDFS	Hadoop Distributed Files System
HP	Hard Pruning
HTTP	Hypertext Transfer Protocol
IaC	Infrastructure-as-Code
ICT	Information and Communications Technology
LSTM	Long Short-Term Memory
ML	Machine Learning
MNIST	Modified National Institute of Standards and Technology
MQTT	Message Queuing Telemetry Transport
NUS	Non-Uniform Sampling
REST	REpresentational State Transfer
RTSP	Real Time Streaming Protocol
SDK	Software Development Kit
SED	Sound Event Detection
SOTA	State Of The Art
SP	Soft Pruning

STS	Site-to-Site
TAD	Text Anomaly Detection
TEE	Trusted Execution Environment
UI	User Interface
VAD	Voice Activity Detection
VCC	Visual Crowd Counting
ViAD	Visual Anomaly Detection
VPN	Virtual Private Network
WP	Work Package
YAML	Ain't Markup Language

Executive Summary

The purpose of this deliverable is to provide the current version of the MARVEL Edge-to-Fog-to-Cloud (E2F2C) framework. The deliverable has been developed within the scope of WP3 of the MARVEL project under Grant Agreement (GA) No. 957337.

The document reports the outcomes of Tasks T3.4 and T3.5. As per the GA, the goals of T3.4 are to:

- Provide the deployment logic that will exploit the full potential of the personalised Federated Learning approach implemented in T3.2;
- Optimise and manage of AI and Deep Learning (DL) component deployment;
- Provide an optimisation strategy, for the component deployment, based on resource requirement and consumption.

Therefore, the first activity of T3.4 was to create the infrastructure that will be used for the deployment of the MARVEL components. This infrastructure consists of a set of hosts part of a Kubernetes cluster. On top of this Kubernetes cluster, MARVdash was placed as a dashboard service for facilitating interaction with the underlying E2F2C testbed, by supplying the landing page for users, allowing them to launch services, design workflows, request resources, and specify other parameters related to execution through a user-friendly interface.

The corresponding goals of T3.5 are to:

- Provide techniques and algorithms for deployment at the edge layer;
- Study the need for compression of AI/DL models based on resource availability;
- Compress such models for reduction of the computational overhead.

Therefore, the first activity of T3.5 was to create a methodology for the compression of Deep Neural Network (DNN) model at training time. Moreover, this methodology was used for the actual compression of the Audio-Visual Crowd Counting (AVCC) model provided by AU.

1 Introduction

One of the objectives of the MARVEL project is to create an Edge-to-Fog-to-Cloud (E2F2C) framework on which a variety of AI/DL components can be deployed in an optimal or almost optimal way. This framework's aim is to realise the deployment logic that will exploit the full potential of the personalised Federated Learning approach implemented in T3.2.

1.1 Purpose and scope

This document reports on the process of creating such an E2F2C execution environment along with a dedicated dashboard for implementing the interaction with the underlying environment, coordinating the execution of the data management platforms and other software components, and mediating external accesses to any service that needs to be exposed outside the MARVEL infrastructure.

Moreover, this document reports on the methodology that was created for the compression of DNN model at training time, along with the actual application of this methodology to the AVCC model provided by AU.

1.2 Relation to other work packages, deliverables, and activities

This document is closely related to all the tasks of WP3. T3.1 deals with the development of AI-based methods for data privacy, the aim of T3.2 is the development of a Federated Learning (FL) framework based on the distribution of data in different locations, and T3.3 tackles the development of multimodal audio-visual AI models. The deployment logic will exploit the full potential of the personalised Federated Learning approach implemented in T3.2. It will be used by the AI MARVEL components that will make use of the AI methods and techniques of the aforementioned tasks.

Moreover, Kubernetes dashboard for the MARVEL E2F2C framework will be used for the deployment of all of the rest MARVEL components, which are in charge of functionalities other than the AI model training and inference. Such components are responsible for data transferring and management. That fact relates this document with i) all the other tasks in different work packages (WPs) that are dedicated to the development of MARVEL components, and ii) all the corresponding deliverables that describe the functionality of the aforementioned components.

Additionally, this document is related to WP5 tasks, since the general aim of the WP is the successful delivery of the MARVEL E2F2C framework, allowing for processing of extreme-scale multimodal data on top of the distributed deployment of the ML models.

1.3 Structure of the report

The document is organised in the following way. Section 2 provides a description of all the MARVEL components that use the MARVdash Kubernetes dashboard to be deployed in one (or more) of the nodes of the corresponding E2F2C cluster. Additionally, this section includes the description of the corresponding Docker images creating process, one of the necessary deployment step. Section 3 focuses on the MARVEL E2F2C framework, explaining the underlying Kubernetes architecture, the containerisation and virtualisation techniques that are used, and the native to Kubernetes tools and techniques that we explored. Moreover, this section includes future plans for the MARVdash component. Section 4 is dedicated to federated learning, mentioning centralised compression methods and efficient federated methods. The compression methods include DynHP and AVCC. Section 5 is dedicated to the description of the corresponding KPIs and to what extent they are fulfilled based on the work carried out until M18 of the project lifetime. Section 6 summarises the conclusions of the deliverable.

2 Available container images of MARVEL components

The backbone of the created MARVEL E2F2C execution environment is a Kubernetes cluster, which allows the deployment of all the MARVEL components on its nodes. The deployment of a component/service in the Kubernetes context takes advantage of the containerisation packaging, creating isolated fully packaged and portable computing environments. This section describes how such execution environments were created for each of the individual MARVEL components, in the form of Docker containers.

2.1 AI subsystem

2.1.1 Visual Anomaly Detection – ViAD

Description:

The ViAD component learns a representation of “normality” from video frames taken from a scene and marks any deviations from this normality as an anomaly. Therefore, an anomaly can be any novel event that has not occurred in the scene before. During inference, the input to this component is a series of video frames from the same scene as the training data, and the output is a flag specifying whether or not there are any anomalies in each frame. Optionally, the component can mark specific areas of the frame, which are containing some kind of anomaly.

Docker Image:

As it can be seen in Listing 1, the base image for the creation of the ViAD docker image is the tensorflow:2.4.1. The following RUN command installs prerequisite software. After that, the main code of the component is copied and installed in the created container. The very last CMD command in the Dockerfile starts the component.

```
FROM tensorflow/tensorflow:2.4.1

RUN apt-key adv --fetch-keys
https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1804/x86_64/3bf86
3cc.pub && \
    apt-get update && \
    apt-get install -y libgl1 libsndfile1 && \
    apt-get install -y ffmpeg && \
    apt-get clean

COPY containers/container_source/mudas-0.2.1-py3-none-any.whl /tmp/
RUN pip install --default-timeout=100 /tmp/mudas-0.2.1-py3-none-any.whl --no-
dependencies

COPY containers/container_source/requirements.txt /tmp/
RUN pip install --default-timeout=100 -r /tmp/requirements.txt

WORKDIR /app
COPY containers/container_source/av_cc_backbone_v10.h5 /app
COPY containers/container_source/utils.py /app/
COPY containers/container_source/source_avcc_vcc.py /app/
COPY containers/avcc/avcc_GRN3_C11/config.py /app/
COPY containers/avcc/avcc_GRN3_C11/output_schema.yaml /app/
```

```
CMD python /app/source_avad_vad.py
```

Listing 1: ViAD Dockerfile

2.1.2 Audio-Visual Anomaly Detection – AVAD

Description:

Similar to ViAD, the AVAD component learns a representation of "normality" from video frames taken from a scene as well as an audio clip from the scene and marks any deviations from this normality as an anomaly. Therefore, an anomaly can be any novel event that has not occurred in the scene before. During inference, the input to this component is a series of video frames from the same scene as the training data as well as the corresponding audio clip, and the output is a flag specifying whether or not there are any anomalies in each frame. Optionally, the component can mark specific areas of the frame, which are containing anomaly, however, the audio clip is not marked for anomalies.

Docker Image:

The structure of the AVAD Dockerfile is basically the same as that of the ViAD component. As we can see in Listing 2, the base image is once again the tensorflow:2.4.1. The first RUN command installs the prerequisite software, while the next commands copy and install the main code of the component. The CMD command at the end of the Dockerfile starts the AVAD component as the last action of the creation of the corresponding container.

```
FROM tensorflow/tensorflow:2.4.1

RUN apt-key adv --fetch-keys
https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1804/x86_64/3bf86
3cc.pub && \
    apt-get update && \
    apt-get install -y libgl1 libsndfile1 && \
    apt-get install -y ffmpeg && \
    apt-get clean

COPY containers/container_source/mudas-0.2.1-py3-none-any.whl /tmp/
RUN pip install --default-timeout=100 /tmp/mudas-0.2.1-py3-none-any.whl --no-
dependencies

COPY containers/container_source/requirements.txt /tmp/
RUN pip install --default-timeout=100 -r /tmp/requirements.txt

WORKDIR /app
COPY containers/container_source/av_cc_backbone_v10.h5 /app
COPY containers/container_source/utils.py /app/
COPY containers/container_source/source_avcc_vcc.py /app/
COPY containers/avcc/avcc_GRN3_C11/config.py /app/
COPY containers/avcc/avcc_GRN3_C11/output_schema.yaml /app/

CMD python /app/source_avad_vad.py
```

Listing 2: AVAD Dockerfile

2.1.3 Visual Crowd Counting – VCC

Description:

The VCC component counts the total number of people present in a given image. Since the annotations in the training data specify the locations of the heads, crowd counting can be viewed as counting the total number of heads present in the image. The input to this component is an image from a scene that may contain people, and the output is a number representing the total count of people in that scene. Optionally, the output may contain a heatmap specifying the density of people for each pixel of the image (also known as “density map”).

Docker Image:

Once again, the structure of the VCC Dockerfile follows the structure of the previous two components. The tensorflow:2.4.1 is the base image. The following commands install the needed libraries and the main code of the component. The last command in the next listing (Listing 3) starts the VCC component itself.

```
FROM tensorflow/tensorflow:2.4.1

RUN apt-key adv --fetch-keys
https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1804/x86_64/3bf86
3cc.pub && \
    apt-get update && \
    apt-get install -y libgl1 libsndfile1 && \
    apt-get install -y ffmpeg && \
    apt-get clean

COPY containers/container_source/mudas-0.2.1-py3-none-any.whl /tmp/
RUN pip install --default-timeout=100 /tmp/mudas-0.2.1-py3-none-any.whl --no-
dependencies

COPY containers/container_source/requirements.txt /tmp/
RUN pip install --default-timeout=100 -r /tmp/requirements.txt

WORKDIR /app
COPY containers/container_source/av_cc_backbone_v10.h5 /app
COPY containers/container_source/utils.py /app/
COPY containers/container_source/source_avcc_vcc.py /app/
COPY containers/vcc/vcc_MT1_C12/config.py /app/
COPY containers/vcc/vcc_MT1_C12/output_schema.yaml /app/

CMD python /app/source_avcc_vcc.py --limit 3
```

Listing 3: VCC Dockerfile

2.1.4 Audio-Visual Crowd Counting AVCC

Description:

Similar to VCC, the AVCC component counts the total number of people present in a given image. However, in the audio-visual case, the input also contains the ambient audio clip taken from the scene. This audio can help improve the accuracy of crowd counting in situations where the image quality is not optimal due to low illumination, occlusion, low resolution or a noisy capture device. Since the annotations in the training data specify the locations of the heads, crowd counting can be viewed as counting the total number of heads present in the image. The input to this component is an image from a scene that may contain people as well as the corresponding audio clip, and the output is a number representing the total count of people in that scene. Optionally, the output may contain a heatmap specifying the density of people for each pixel of the image (also known as "density map").

Docker Image:

AVCC is the last one in a series of MARVEL components with similar Dockerfiles. AVCC Dockerfile is depicted in Listing 4 below.

```
FROM tensorflow/tensorflow:2.4.1

RUN apt-key adv --fetch-keys
https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1804/x86_64/3bf86
3cc.pub && \
    apt-get update && \
    apt-get install -y libgl1 libsndfile1 && \
    apt-get install -y ffmpeg && \
    apt-get clean

COPY containers/container_source/mudas-0.2.1-py3-none-any.whl /tmp/
RUN pip install --default-timeout=100 /tmp/mudas-0.2.1-py3-none-any.whl --no-
dependencies

COPY containers/container_source/requirements.txt /tmp/
RUN pip install --default-timeout=100 -r /tmp/requirements.txt

WORKDIR /app
COPY containers/container_source/av_cc_backbone_v10.h5 /app
COPY containers/container_source/utils.py /app/
COPY containers/container_source/source_avcc_vcc.py /app/
COPY containers/avcc/avcc_UNS_F1/config.py /app/
COPY containers/avcc/avcc_UNS_F1/output_schema.yaml /app/

CMD python /app/source_avcc_vcc.py --limit 3
```

Listing 4: AVCC Dockerfile

2.1.5 Sound Event Detection – SED

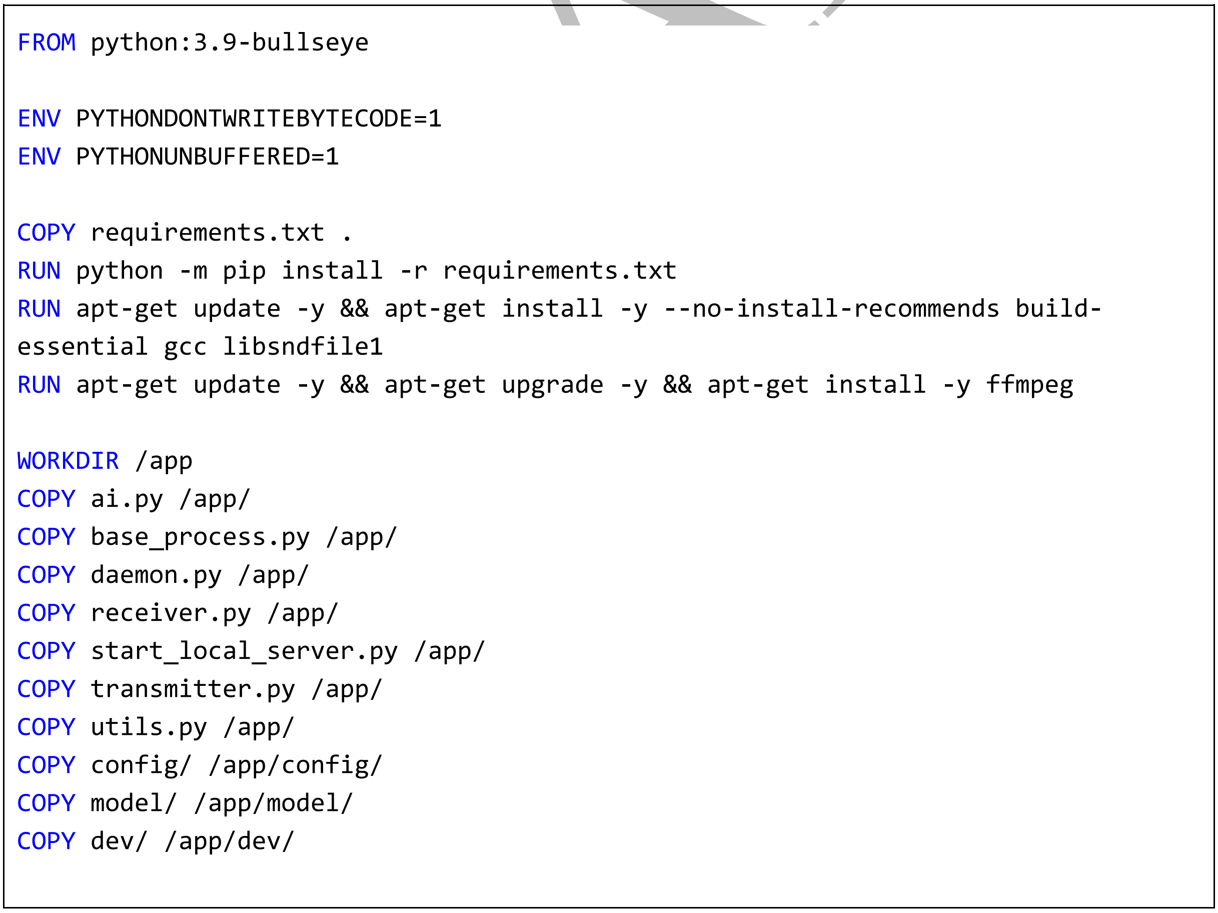
Description:

The SED component provides the detection of characteristic sounds in short time units. A characteristic sound is a sound that can be described by a specific label, i.e., a sound event. This functionality is used in the various use cases of MARVEL to offer the ability to detect actions and events through sound. The specific sound events will be dependent on the use cases and the detection of the sound events can be used as standalone information or as complementary information to other systems. A SED component takes as an input an audio signal and provides detection of sound events in pre-specified units of time. The component is developed to run on powerful computing nodes: either with a high-performance Central Processing Unit (CPU) or with a computer system having a Graphics Processing/Processor Unit (GPU) available. The component is designed to operate in real-time while consuming real-time audio-visual streams.

Docker Image:

In Listing 5, we report how to build the Docker Image of the current version of the SED component. Note that the docker image can evolve along with the MARVEL project with new functionalities.

Python:3.9-bullseye is the chosen base image. The ENV commands define some environmental variables in the newly created Docker container. The following set of commands installs necessary libraries for the functionality of the component. The second of the last command creates a dedicated user, while the USER command switches to that user. Finally, the CMD command starts the SED component.



```
FROM python:3.9-bullseye

ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1

COPY requirements.txt .
RUN python -m pip install -r requirements.txt
RUN apt-get update -y && apt-get install -y --no-install-recommends build-
essential gcc libsndfile1
RUN apt-get update -y && apt-get upgrade -y && apt-get install -y ffmpeg

WORKDIR /app
COPY ai.py /app/
COPY base_process.py /app/
COPY daemon.py /app/
COPY receiver.py /app/
COPY start_local_server.py /app/
COPY transmitter.py /app/
COPY utils.py /app/
COPY config/ /app/config/
COPY model/ /app/model/
COPY dev/ /app/dev/
```

```
RUN adduser -u 5678 --disabled-password --gecos "" appuser && chown -R appuser
/app
USER appuser
CMD ["python", "daemon.py"]
```

Listing 5: Sound Event Detection Dockerfile

2.1.6 Audio Tagging - AT

Description:

The AT component provides information about the activity of characteristic sounds inside audio segments with predefined fixed lengths. This functionality is used in the various use cases of MARVEL to offer the ability to recognise sounds related to actions or events with coarse time resolution. The specific sound classes recognised will be dependent on the use cases and the recognised sound class activity can be used as standalone information or as complementary information to other systems. The component uses the same code base as the SED component.

Docker Image:

The Docker image of AT is created using the same Dockerfile that SED uses (see 2.1.5).

```
FROM python:3.9-bullseye

ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1

COPY requirements.txt .
RUN python -m pip install -r requirements.txt
RUN apt-get update -y && apt-get install -y --no-install-recommends build-
essential gcc libsndfile1
RUN apt-get update -y && apt-get upgrade -y && apt-get install -y ffmpeg

WORKDIR /app
COPY ai.py /app/
COPY base_process.py /app/
COPY daemon.py /app/
COPY receiver.py /app/
COPY start_local_server.py /app/
COPY transmitter.py /app/
COPY utils.py /app/
COPY config/ /app/config/
COPY model/ /app/model/
COPY dev/ /app/dev/

RUN adduser -u 5678 --disabled-password --gecos "" appuser && chown -R appuser
/app
USER appuser
CMD ["python", "daemon.py"]
```

Listing 6: Acoustic Scene Classification Dockerfile

2.1.7 CATFlow

Description:

CATFlow is a software asset developed by GRN, where the input is a video stream and the output is a list of traffic objects tracked over the camera field of view. CATFlow classifies vehicles into six different classes: car, bus, light goods vehicles, heavy good vehicles, bicycle, and motorcycle. In addition, each object (e.g., vehicle or pedestrian) is tracked and its trajectory extracted and stored for visualisation or further processing.

In the current implementation, the Real-Time Streaming Protocol (RTSP) streaming protocol is used to receive a video stream and each frame is grabbed using FFMPEG. Each frame is sequentially fed first into an object detector (YOLO4 trained on specific traffic object classes provided by the transport authorities) and then a multi-object tracker (MOSSE) to track traffic entities moving across the scene. Geometry-based algorithms are used to compute variables of interest such as vehicle speed over a predetermined trajectory. Pedestrians on the other hand are handled differently since they often follow a random path thus speed cannot be calculated using these geometrical methods.

To safeguard GRN's IP on the CATFlow software asset, the CATFlow configurator was uploaded as part of the MARVEL registry. This configurator is then responsible for pulling the CATFlow image onto the device from GRN's Azure registry.

Docker Image:

The CATFlow image itself is split into 2 files: a Dockerfile-base and a Dockerfile. The Dockerfile-base will build an image consisting of all the libraries necessary to run CATFlow (i.e., ffmpeg, CUDNN, python, OpenCV, etc). The parent image is NVIDIA's cuda image. This was done in order to reduce the build time for the CATFlow image since these libraries rarely change. After building the base image, the code is added. The code is written in Python but compiled into Cython. These steps - excluding the base image build - are handled automatically as part of GRN's CI/CD pipeline. Hence, the image is always up-to-date.

The Configurator image follows the same idea of being split into 2 parts. The code itself is not compiled to Cython – however, this is subject to change in future releases. The Configurator will pull the built CATFlow image from GRN's repository using Docker commands. Similarly, the image is always up-to-date as it follows the same CI/CD pipeline.

The images themselves are built on a Ubuntu 20.04 system. CUDA 11.2.1, CUDNN 8 and Python 3.9 are being used along with OpenCV 4.5.2. The compute capability of the GPU under use by CATFlow needs to be addressed as this will affect the base image. This is due to the CUDA architectures compiled by OpenCV when building the image. The corresponding Dockerfile is not presented in this document, as it does for other MARVEL components, due to privacy reasons. We want to avoid exposing any sensitive information regarding the component itself or the sources of its input.

2.1.8 Text Anomaly Detection - TAD

Description:

TAD is a component that automatically detects anomalous events in data, for example, anomalous vehicle velocities and trajectories. TAD takes as input the JSON messages outputted from CATFlow and after processing flags any anomalous behaviour. The TAD component also requires the storage and access to a Dataset of CATFlow outputs such that a model of the normal behaviour on the scene being observed is developed and updated. The current TAD version considers the speed of vehicles and flags anomalous low or high values.

The current implementation of the TAD component makes use of the CATFlow output, specifically the vehicle speed calculation. In addition, TAD accesses the GRN CATFlow database to model the vehicle speed normally observed on the road segment being monitored. The TAD can also use a preloaded model instead of accessing the database, such that the anomaly detection tool can still be used in the event of the database being inaccessible. During the last step, TAD performs a z-score test for any new vehicle speed value extracted from the CATFlow output and if the new data point is not within the range of the Z-score test, the TAD flags or raises an alarm to indicate the occurrence of the anomalous event.

In the current implementation, two types of anomalies can be detected; (a) anomalously low speeds, and (b) anomalously high speeds. Anomalously low speeds usually indicate that either a vehicle has stopped moving, possibly creating an obstruction or unusual traffic jams. Anomalously high-speed events usually indicate vehicles that are over speeding (velocity beyond sign posted limit) and are useful in estimating the safety of the road segment under observation.

Docker Image:

The corresponding TAD image is built on an Ubuntu 20.04 system with python 3.9. The packages required are installed through the docker file. Listing 7 shows the commands for the manual deployment of TAD using the created Docker image.

```
docker login registry.marvel-platform.eu

docker pull registry.marvel-platform.eu/tad:0

docker run -it registry.marvel-platform.eu/tad:0 /bin/sh
```

Listing 7: TAD container creation commands

The first command logins the user in the MARVEL platform registry. A pull command follows that fetches the available image, while the run command creates the corresponding container from the downloaded image. As a result, a TAD container is created and started.

2.2 Security, privacy and data protection subsystem

2.2.1 EdgeSec Virtual Private Network (VPN)

Description:

EdgeSec VPN adopts the n2n architecture, according to which there are two key components: edge and Super nodes. The edge nodes use the Super Nodes for discovering other edge nodes. The Super Nodes are also used for routing the traffic when the nodes are behind symmetrical firewalls. The n2n, and therefore the EdgeSec VPN, is a peer-to-peer VPN that works on the second layer of the OSI model, allowing the peers to maintain reachability across NATs and firewalls. Edge nodes that participate in the same virtual network form a community. Super Nodes are able to serve more than one community and a single computer can join multiple communities.

Docker Image:

The Docker Image of the EdgeSec VPN is created using the Dockerfile depicted in Listing 8. As it can be seen, the base image that is used is the one for Ubuntu 18.04. Some prerequisite software is installed according to the first lines of the Dockerfile. Then, the main code of the component is copied and installed in the container. The port 4194 is exposed to the outside world, while the last command starts the component.

```
FROM ubuntu:18.04
RUN apt-get update && \
        apt-get install -y build-essential net-tools autoconf pkg-config
RUN mkdir -p /usr/ipsec

WORKDIR /usr/ipsec

COPY ./ .
RUN ./autogen.sh
RUN ./configure
RUN make
RUN make install

EXPOSE 4194/udp

CMD ["sh", "init_script.sh"]
```

Listing 8: EdgeSec VPN Dockerfile

2.2.2 EdgeSec Trusted Execution Environment (TEE)

Description:

EdgeSec TEE requires an Intel-SGX enabled machine and the installation of the Docker software. Each application that is secured with EdgeSec TEE lays on top of a machine that supports Intel SGX. In order to use EdgeSec TEE and take full advantage of the security characteristics that it offers, an application developer needs to follow these steps:

(i) get access to infrastructure that is Intel SGX-enabled,
(ii) download the EdgeSec TEE's docker image that is uploaded to the MARVEL registry by FORTH,
(iii) launch the EdgeSec TEE container from this image,
(iv) copy the python application inside the container's file system and install any required python library or package,
(v) execute the python application that is secured by SCONE during the total execution time.

Docker Image:

Once downloaded from the MARVEL docker image registry, the EdgeSec TEE component can be deployed by the following certain steps:

```
docker login registry.marvel-platform.eu

docker pull registry.marvel-platform.eu/docker-sgx:0

docker run -it registry.marvel-platform.eu/docker-sgx:0 /bin/sh
```

Listing 9: EdgeSec container creation commands

As a result, an EdgeSec TEE container is created and started. The Python version that is supported by EdgeSec TEE is 3.7.3 (in an environment of Alpine Linux 3.10). It is possible to install packages and libraries within the container. After the successful installation of Python libraries, "normal" operation of Python programmes is enabled as in traditional setups (i.e., without the support of Intel SGX, SCONE and EdgeSec TEE). Within the container of EdgeSec TEE, however, the Python application is executed within Intel SGX enclaves, which offer security, code integrity and data confidentiality. Examples for EdgeSec TEE execution and libraries installation are available in Deliverable 4.2.

2.2.3 VideoAnony

Description:

VideoAnony aims to anonymise the detected faces and car plates from raw video feeds coming from the Closed-Circuit Television (CCTV) cameras from each pilot site. The anonymisation is performed via image redaction methods, starting from classic image processing techniques, such as blurring, towards the more advanced GAN-based face-swapping techniques, which is under development within the MARVEL project. Current VideoAnony component receives the incoming raw video stream either via RTSP or direct cable access. It then employs the yolov5 detector for face and car plate detection which is finetuned with related public dataset and pilot-provided annotations. With the detected regions of interest, the component finally blurs them. In addition, we are also developing a lighter version of the advanced GAN-based face swapping model based on state-of-the-art methods. In the early phase, we specifically addressed the challenges of pose preservation and varying size of the detected faces from CCTV videos, while in the current phase we are focusing on reducing the computational complexity of the model. The ongoing efforts are mainly guided by a couple of directions, i.e., components replacement and weight quantisation, whose results are not yet conclusive at the moment.

Docker Image:

In Listing 10, we report how to build the Docker image (i.e., Dockerfile) of the current version of VideoAnony. Note that the docker image will be evolved along with the MARVEL project with new functionalities included, e.g., streaming output, and incorporating face-swapping for anonymisation.

```
# Start FROM Nvidia PyTorch image
https://ngc.nvidia.com/catalog/containers/nvidia:pytorch
FROM nvcr.io/nvidia/pytorch:21.10-py3

ARG USER=standard
ARG USER_ID=1000 # uid from the previus step
ARG USER_GROUP=standard
ARG USER_GROUP_ID=1000 # gid from the previus step
ARG USER_HOME=/home/${USER}
# create a user group and a user (this works only for debian based images)
RUN groupadd --gid $USER_GROUP_ID $USER \
    && useradd --uid $USER_ID --gid $USER_GROUP_ID -m $USER

# Install linux packages
RUN apt-get update && apt-get upgrade -y
RUN \
    DEBIAN_FRONTEND=noninteractive apt-get install -y libgl1-mesa-glx libsm6
libxext6 libxrender-dev libglib2.0-0

# Install python dependencies
```

```
COPY requirements.txt .
RUN python -m pip install --upgrade pip
RUN pip uninstall -y torch torchvision torchtext
RUN pip install --no-cache -r requirements.txt \
    torch==1.11.0+cu113 torchvision==0.12.0+cu113 -f
https://download.pytorch.org/whl/cu113/torch_stable.html

# Install base dependencies + gstreamer
# RUN pip uninstall -y opencv-python
RUN apt-get update

RUN DEBIAN_FRONTEND=noninteractive apt-get -y install ffmpeg

RUN \
    DEBIAN_FRONTEND=noninteractive \
    apt-get -y install build-essential \
    cmake \
    pkg-config \
    libgtk-3-dev \
    libavcodec-dev \
    libavformat-dev \
    libswscale-dev \
    libv4l-dev \
    libxvidcore-dev \
    libx264-dev \
    libjpeg-dev \
    libpng-dev \
    libtiff-dev \
    gfortran \
    openexr \
    libatlas-base-dev \
    python3-dev \
    python3-numpy \
    libtbb2 \
    libtbb-dev \
    libdc1394-22-dev

RUN \
    DEBIAN_FRONTEND=noninteractive \
    apt-get install -y libgstreamer1.0-0 \
    gstreamer1.0-plugins-base \
    gstreamer1.0-plugins-good \
    gstreamer1.0-plugins-bad \
    gstreamer1.0-plugins-ugly \
    gstreamer1.0-libav \
    gstreamer1.0-doc \
    gstreamer1.0-tools \
    gstreamer1.0-x \
    gstreamer1.0-alsa \
    gstreamer1.0-gl \
    gstreamer1.0-gtk3 \
    gstreamer1.0-qt5 \
    gstreamer1.0-pulseaudio \
    gstreamer1.0-rtsp \
    libgstreamer1.0-dev \
    libgstreamer-plugins-base1.0-dev \
    cmake \
    protobuf-compiler \
    libgtk2.0-dev \
    ocl-icd-opencl-dev

# Clone OpenCV repo
WORKDIR /
RUN git clone https://github.com/opencv/opencv.git
WORKDIR /opencv
RUN git checkout 4.5.4

# # Build OpenCV
RUN mkdir /opencv/build
WORKDIR /opencv/build
RUN ln -s /opt/conda/lib/python3.8/site-packages/numpy/core/include/numpy
/usr/include/numpy
RUN cmake -D CMAKE_BUILD_TYPE=RELEASE \
    -D INSTALL_PYTHON_EXAMPLES=ON \
    -D INSTALL_C_EXAMPLES=OFF \
    -D PYTHON_EXECUTABLE=$(which python) \
    -D BUILD_opencv_python2=OFF \
    -D CMAKE_INSTALL_PREFIX=$(python -c "import sys; print(sys.prefix)") \
    -D PYTHON3_EXECUTABLE=$(which python3) \
    -D PYTHON3_INCLUDE_DIR=$(python -c "from distutils.sysconfig import
get_python_inc; print(get_python_inc())") \
    -D PYTHON3_PACKAGES_PATH=$(python -c "from distutils.sysconfig import
get_python_lib; print(get_python_lib())") \
    -D WITH_FFMPEG=ON \
    -D WITH_GSTREAMER=ON \
    -D BUILD_EXAMPLES=ON ..
RUN make -j$(nproc)

# Install OpenCV
RUN make install
RUN ldconfig

# Create working directory
RUN mkdir -p /app
WORKDIR /app
```

```
# set container user
USER $USER
```

Listing 10: VideoAnony Dockerfile

Then the init_script.sh invokes the python code for the main script of video anonymisation, where you will provide the source file (either a live stream or a video file) to be anonymised. One example is provided as follows:

```
#!/bin/sh
python src/anonymize.py --source data/videos
```

Listing 11: Initialisation script for VideoAnony

2.2.4 AudioAnony

Description:

The AudioAnony component is based on a basic signal processing technique and generates a new waveform applying a modification of the associated poles, computed from the LPC coefficients on a frame basis; as such, the related formant positions are moved, modifying the spectral envelope and therefore the voice characteristics while the speech content is preserved. The formant shifting is controlled by a single parameter, the so-called McAdams coefficient. This component operates together with the VAD module so that waveform modifications only take place when speech is detected. In absence of speech, the unmodified waveform is transmitted instead.

Docker Image:

In Listing 12, we report how to build the example Docker image (i.e., Dockerfile):

```
FROM python:3.8-slim-buster
RUN apt-get update && apt-get -y install libsndfile-dev
COPY requirements.txt .
RUN python -m pip install --upgrade pip
RUN pip install --no-cache-dir -r requirements.txt
RUN mkdir -p /app
WORKDIR /app
# Copy contents
COPY . /app
CMD ["bash", "init_script.sh"]
```

Listing 12: AudioAnony Dockerfile

The init_script.sh invokes the python code for the conversion process with the proper arguments according to the availability of an accompanying segmentation file:

```
audio_dir=/app/data/audio
target_dir=/app/runs
sfx="_conv.wav"
wavL=`ls -1 ${audio_dir}/*.wav 2>/dev/null`
```

```
if [ -z ${wavL} ]; then
    echo "no wav files found in ${audio_dir}"
    exit
else
    for wav in ${wavL}; do
    id=`basename ${wav} .wav`
    wav_out=${target_dir}/${id}${sfx}
    seg=`dirname $wav`/${id}.json
    if test -f ${seg}; then
        echo "converting ${wav} to ${wav_out} using ${seg}"
        python src/conversion.py --filename $wav --filenameout $wav_out --coeff
-1 --seg ${seg}
    else
        echo "converting ${wav} to ${wav_out}"
        python src/conversion.py --filename $wav --filenameout $wav_out --coeff
-1
    fi
    done
fi

conversion.py accepts the following arguments:
--filename WAVFILE file to be processed
--seg  SEG optional json file with speech segmentation
--filenameout WAVFILE name of the output file
--coeff FLOAT conversion coefficient; if -1 the coefficient is chosen randomly
between 0.8 and 1.2
```

Listing 13: Initialisation script for AudioAnony

2.2.5 VAD (devAIce)

Description:

devAIce is a Software Development Kit (SDK) written in C++, and represents AUD’s modular technology that wraps all its AI technologies for intelligent audio analytics, including the award-winning openSMILE, an audio features extraction tool in a high-dimensional space. devAIce is optimised to run on powerful computing nodes (GPUs, CPUs) but also on high-end edge devices (Raspberry Pi). Furthermore, another trimmed-down version containing only openSMILE toolkit can be extracted if needed and can be deployed on edge devices with very limited computational resources, although no use case currently requires it.

devAIce exposes multiple interfaces in Python, iOS, Android, and C. Although this is not a limitation where it cannot be used with different programming languages and platforms, however, in that case, the end user has to manually build his own wrapper around the C interface, as it has done to provide the Python, iOS and Android.

As mentioned previously, devAIce contains multiple AI subsystems, each function in its own way and needs to be configured separately. One of those subsystems is the Voice Activity Detection (VAD) module. Being the module of interest in R1, the AI model that has been trained to output frame-level predictions. A second layer, purely algorithmic, exploits those frame-level predictions and aggregates them based on certain conditions and parameters, in order to provide the speech segments boundaries as final output. This model adopts a feed-forward Long Short-Term Memory (LSTM) architecture with attention and has been trained on a very large amount of artificially mixed data. During the project, the model has been retrained with a recent state-of-the-art architecture to also detect music segments within the audio sequence.

Docker Image:

In the first stages, ongoing discussions were taking place to decide the efficient way to deploy devAIce VAD. Being a key component in the audio anonymisation pipeline along with AudioAnony, it will detect the speech segments from an ongoing audio stream, which will be anonymised before being forwarded to the next layers for further analysis (fog and cloud). It is decided now that VAD+AudioAnony will construct together a new AV source, which consumes the AV streams from the pilot’s microphones and exposes the anonymised version to the other AI components (e.g., SED). As a result, VAD and AudioAnony will be fused and referred to as a single composite component for resource and performance-wise reasons that have been tackled in detail during the focused meetings. Both components will then be part of the same docker image, which will be implemented and deployed in R1 on the edge (on Raspberry pies in the case of MT network).

However, a first version of the docker image, containing only VAD has been published before a decision has been taken to combine VAD and AudioAnony. The image initialises the python virtual environment responsible for running the VAD module and runs devAIce Command Line Interface (CLI) on a test wav file to output speech segments. The contents of the docker file used to create the image are listed in Listing 14 below, however, this image will not be used in the later stages of the project.

```
FROM python:3.8
RUN apt-get update && apt-get -y install libsndfile-dev
COPY requirements.txt .
COPY devAIce-SDK-3.4.0-2022-02-23 .
RUN python -m pip install --upgrade pip
RUN pip install --no-cache-dir -r requirements.txt
RUN pip install devAIce-SDK-3.4.0-2022-02-23/bin/python/devaice-3.4.0-py3-none-
linux_x86_64.whl
RUN mkdir -p ./sample
WORKDIR ./sample
# Copy test wav file and run vad on CLI
COPY test.wav .
CMD ["bash", “../devAIce-SDK-3.4.0-2022-02-23/bin/linux-x86_64/devaice-sdk-cli –
vad –resource_root=../devAIce-SDK-3.4.0-2022-02-23/res test.wav”]
```

Listing 14: Voice Activity Detection Dockerfile

However, the command being run in the docker file is just for test purposes and is not suitable for our real-time use cases. Therefore, for this purpose, a python script has been set. The script ingests audio stream using PyAudio, applies devAIce VAD module in a separate thread, and devAIce VAD+Music model in another thread. The first thread goal is to detect speech segments, anonymise them and store them in memory. If no speech is detected, the audio is stored in memory as it is. The other thread outputs the boundaries of the speech and music segments, with stdout being the terminal console. At the end of the script, when the stream is manually closed, the collected anonymised audio will be exported as a wav file for integrity check. This same script will be adapted to connect with the pilot's microphones and cameras, and the collected audio instead of being exported will be forwarded through RTSP. With regards to speech and music segments, the inference results will be forwarded to the adequate MQTT topic.

This described script exists under the name "rtsp_audio_test_w_vad.py", and accepts as parameter the resource path of the devAIce SDK as -r/--resource_path, which is a required argument that contains the path to the folder containing the legitimate devAIce license file.

2.3 Data management and distribution subsystem

2.3.1 StreamHandler

Description:

INTRA's StreamHandler Platform is a distributed streaming platform for handling real-time data based on Apache Kafka. It can efficiently ingest and handle massive amounts of data into processing pipelines, both for real-time and batch processing. The platform and its underlying technologies can support any type of data-intensive Information and Communications Technology (ICT) services (Artificial Intelligence, Business Intelligence, etc.) in different environments, from cloud to edge.

In the context of MARVEL, StreamHandler contributes with a newly developed module that aims at providing audio-visual data management capabilities. To that end, StreamHandler:

- Receives and efficiently archives live streams of audio-visual binary data from all relevant MARVEL sensors, devices, and components during system operation;
- Provides access to archived audio-visual binary data to the MARVEL UI (SmartViz) by editing and compiling bespoke audio-visual data upon demand, that correspond to a specific anomaly/event detected by the AI components.;
- Supports the expansion of the data set of the DataCorpus by relaying selected archived audiovisual data to it.

The StreamHandler AV module is based on Python 3[1], rtsp-simple-server[2], minIO[3], FastAPI[4], and Docker[5] technologies.

Docker Image:

Concerning MARVEL's first integrated version, the deployment of StreamHandler will take place on the fog servers of each pilot, so that only selections of data are propagated to the cloud. However, in the future, StreamHandler might also be deployed on edge and/or cloud layers as dimmed necessary. All of its instances will be controlled by the Kubernetes environment provided by MARVdash.

The component consists of the following services:

- StreamHandlerAPI
- minioscripting
- minioserver

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

[2] https://github.com/aler9/rtsp-simple-server

[3] https://min.io/

[4] https://fastapi.tiangolo.com/

[5] https://www.docker.com/

- rtspserver
- ffmpeg

Below, in Listing 15, the corresponding Ain't Markup Language (YAML) files are depicted.

```
FROM python:3.10-slim-bullseye
RUN apt-get update
RUN apt-get install -y ffmpeg
WORKDIR /code
COPY ./requirements.txt /code/requirements.txt
RUN pip install --no-cache-dir --upgrade -r /code/requirements.txt
COPY ./app /code/app
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "80"]

#ffmpeg dockerfile
FROM ubuntu:latest
RUN apt-get update && apt-get install -y ffmpeg

#minioserver docker-compose
version: "3.9"  # optional since v1.27.0
services:
  minio:
    image: minio/minio:latest
    ports:
      - "9001:9001"
      - "9000:9000"
    environment:
      - MINIO_ROOT_USER=minio
      - MINIO_ROOT_PASSWORD=minio123
    command: "server --console-address :9001 /data "
    volumes:
      - ../minio_data/:/data
    networks:
      - rtsp-server_marvel-network
networks:
  rtsp-server_marvel-network:
    external: true


#minio-scripting docker-compose
version: "3.9"  # optional since v1.27.0
services:
  minio:
    image: minio/mc
    entrypoint: bash  -x /tmp/minio_client_script.sh
    volumes:
      - ../segmentation_data/:/data
      - ./:/tmp/
    networks:
      - rtsp-server_marvel-network
networks:
  rtsp-server_marvel-network:
    external: true


#minio-server docker-compose
version: "3.9"  # optional since v1.27.0
services:
  minio:
    image: minio/minio:latest
    ports:
      - "9001:9001"
      - "9000:9000"
    environment:
      - MINIO_ROOT_USER=minio
      - MINIO_ROOT_PASSWORD=minio123
    command: "server --console-address :9001 /data "
    volumes:
      - ../minio_data/:/data
    networks:
      - rtsp-server_marvel-network
networks:
  rtsp-server_marvel-network:
    external: true
```

Listing 15: StreamHandler Dockerfile

For more information about StreamHandler the reader is referred to deliverable D2.2 - Management and distribution Toolkit – initial version.

2.3.2 Data Fusion Bus - DFB

Description:

The DFB is a customisable component that implements a trustworthy way of transferring large volumes of heterogeneous data between several connected components and the permanent storage. It comprises a collection of dockerised, open-source components which allow easy deployment and configuration as needed.

DFB’s architectural design addresses several challenges that are raised by both the large volume and the heterogeneous nature of data from different sources, taking into consideration the needs and restrictions of the employed components. The main addressed challenges include:

- seamless aggregation of data with different structures or formats;
- a cluttering threat to the components due to the quantity of the input data;
- access to data through a common, safe, easy-to-consume interface.

Inherent to DFBs design is the efficient handling of the enormous volume of the data that need storage and manipulation, as well as mechanisms to remediate potential bottlenecks, lag, or high network load. These design decisions enable horizontal scalability while providing a solution that is cloud-native with stateless components capable of being flexibly deployed. DFB follows the middleware approach by aligning data streams for time and granularity and creating a User Interface (UI) that serves as the interface of the platform, customised to aggregate multiple streams, thereby allowing seamless service of data to the network analysis and visualisation.

The key capabilities of DFB are:

- Data aggregation from heterogeneous data sources and data stores.
- Real-time analytics, offering ready-to-use ML algorithms for classification, clustering, regression, and anomaly detection.
- An extendable and highly customisable UI for Data Analytics, manipulation, and filtering, as well as functionality for managing the platform.
- Web Services for exploiting the platform outputs for Decision Support.
- Applications for Smart Production, Digitisation, and IoT, among others.

In the context of the initial version of the MARVEL integrated framework, the DFB deployment was designed to meet the requirements of the respective use cases that were defined for this release. To that end, the DFB was foreseen to be deployed at the cloud layer and specifically at the OpenStack infrastructure node managed by PSNC. Three instances of the DFB are configured to run for increased reliability and performance.

The DFB is composed of a set of different services that were deployed through MARVdash. For each service, a container image and an associated YAML configuration template document were uploaded to MARVdash using its web interface. The deployed DFB services are:

- Kafka. Distributed streaming platform for managing the streams of inference results produced by AI components in MARVEL. Three instances of this service were deployed.
- DataFusion connector. DataFusion connector for the ElasticSearch search engine for transferring information published on Kafka topics for persistent storage.
- ElasticSearch is a distributed, multitenant-capable, full-text search engine used for the persistent storage of all incoming inference results in MARVEL. This service also includes Kibana, a free and open user interface that allows visualisation of ElasticSearch data.
- DataFusion es-proxy. DataFusion es-proxy is Elasticsearch search engine for exposing a REpresentational State Transfer (REST) Application Programming Interface (API) to access the Elasticsearch data and perform queries (used by SmartViz).
- Prometheus. Prometheus event monitoring and alerting platform, configured for key measurements related to the Kafka operation and performance.
- Jmx-exporter. JMX service for scraping a kafka broker and exporting related metrics to Prometheus. One instance is deployed per Kafka service.
- Grafana. Interactive visualisation web application for the metrics aggregated at Prometheus.

Docker Image:

Listing 16 below reports an indicative YAML file that was used for the deployment configuration of the core Kafka service image through MARVdash.

```
# kafka.template.yaml
```

```
apiVersion: v1
kind: Service
metadata:
  name: ${NAME}
spec:
  ports:
  - name: broker
    port: 9092
  selector:
    app: cp-kafka
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: cp-kafka
spec:
  podManagementPolicy: OrderedReady
  replicas: ${SERVERS}
  serviceName: ${NAME}
  selector:
    matchLabels:
      app: cp-kafka
  template:
    metadata:
      labels:
        app: cp-kafka
    spec:
      # affinity:
      #   podAntiAffinity:
      #     preferredDuringSchedulingIgnoredDuringExecution:
      #     - podAffinityTerm:
      #         labelSelector:
      #           matchExpressions:
      #           - key: app
      #             operator: In
      #             values:
      #             - cp-kafka
      #         topologyKey: kubernetes.io/hostname
      #       weight: 1
      containers:
      - command:
        - sh
        - -c
        - |
          export KAFKA_BROKER_ID=${HOSTNAME##*-} && \
          export
KAFKA_ADVERTISED_LISTENERS=PLAINTEXT://${HOSTNAME}.${NAME}.${NAMESPACE}.svc:9092
,EXTERNAL://${HOST_IP}:$((31090 + ${KAFKA_BROKER_ID})) && \
          mkdir -p ${PRIVATE_DIR}/.cp-kafka-$KAFKA_BROKER_ID/data && \
          find /etc -type f -exec sed -i "s|/var/lib/kafka/*|${PRIVATE_DIR}/.cp-
kafka-$KAFKA_BROKER_ID/|" {} \; && \
          unset KAFKA_HOST; unset KAFKA_PORT && \
          /etc/confluent/docker/run
        env:
        - name: HOST_IP
          valueFrom:
            fieldRef:
              fieldPath: status.hostIP
        - name: KAFKA_HEAP_OPTS
          value: -Xmx1G -Xms1G
        - name: KAFKA_ZOOKEEPER_CONNECT
          value: ${ZOOKEEPER}
        # - name: KAFKA_METRIC_REPORTERS
        #   value: "io.confluent.metrics.reporter.ConfluentMetricsReporter"
        - name: CONFLUENT_METRICS_REPORTER_BOOTSTRAP_SERVERS
          value: PLAINTEXT://${NAME}:9092
        - name: KAFKA_LISTENER_SECURITY_PROTOCOL_MAP
          value: PLAINTEXT:PLAINTEXT,EXTERNAL:PLAINTEXT
        # - name: KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR
        #   value: "3"
        - name: KAFKA_JMX_PORT
          value: "5555"
        - name: KAFKA_AUTO_CREATE_TOPICS_ENABLE
          value: "true"
        - name: KAFKA_LOG_RETENTION_BYTES
          value: '5368709120'
        - name: KAFKA_LOG_RETENTION_MS
          value: '7200000'
        image: confluentinc/cp-kafka:5.3.1
        imagePullPolicy: IfNotPresent
        name: cp-kafka-broker
        ports:
        - containerPort: 9092
          name: broker
  updateStrategy:
    type: RollingUpdate
---
kind: Template
name: Kafka with custom retention
description: Distributed streaming platform. log.retention.bytes = 5GB,
log.retention.ms = 2 hours
singleton: yes
datasets: no
variables:
- name: NAMESPACE
  default: default
```

```
- name: NAME
  default: kafka
- name: SERVERS
  default: 3
  help: Set to an odd number from 3 and above
- name: ZOOKEEPER
  default: zookeeper
  help: ZooKeeper service name
- name: PRIVATE_DIR
  default: /private
```

Listing 16: Data Fusion Bus Dockerfile

2.3.3 DatAna

Description:

DatAna is one of the Data Management Platforms dealing with the ingestion, transformation and routing of the inference results provided by the AI subsystem components towards the DFB for further visualisation or connection to the MARVEL Data Corpus. As such, it plays an important role in the interconnection of the inference models results and the MARVEL system throughout the computing continuum (E2F2C).

DatAna is based in the Apache NiFi ecosystem (Apache NiFi, NiFi Registry and Apache MiNiFi). In MARVEL, DatAna interfaces with the inference results from the AI subsystem via dedicated message brokers (MQTT) deployed in the different layers of the system. Each of the AI inference modules working over an AV stream produces a message in a dedicated topic with the results in MQTT. The nearest DatAna instance in the layer, which is the NiFi or MiNiFi service deployed in the same infrastructure as the MQTT, subscribes to these topics, retrieves the messages and further processes them to comply with the specific models of Alerts, Anomalies or MediaEvents defined as unified data models for the project. Therefore, the integration of the system is loosely coupled and facilitates the interaction among components.

As hinted in the previous paragraphs, the deployment of DatAna in MARVEL is done in multiple layers and instances and comprises several tools. In particular, DatAna requires deployment of at least one instance per layer and use case. This means that at least one instance of Apache NiFi must be deployed at the MARVEL cloud (NiFi master), and one instance must be deployed in each of the fog servers of the pilots (one instance per pilot). Each of the fog NiFi instances at the fog must enable communication with the NiFi master in the cloud. Similarly, several MiNiFi agents (instances) might be developed at the edge devices controlled by the Kubernetes environment.

Docker Image:

In the case of Apache NiFi, the docker image is inherited from the v1.15.3 version of Apache NiFi provided in the docker hub. For the deployment of NiFi in MARVdash, a helm chart of NiFi based on the cetic helm chart[6] has been provided and adapted. A YAML file for NiFi has been tailored and uploaded to MARVdash for each of the instances of NiFi to be deployed for each architecture layer and pilot, including some properties to be populated to indicate the

[6] https://www.cetic.be/

deployment infrastructure, location of the internal NiFi repositories, etc. The YAML is not provided here, as it is more than 1200 lines long.

For MiNiFi, a similar approach has been used. The official docker image for x86/64-based systems corresponds to the one downloadable from docker hub. For ARM-based systems (i.e., Raspberry Pi) a dedicated docker image of MiNiFi has been uploaded to MARVdash. Dedicated YAML files for the deployment in MARVdash have been also produced and uploaded.

To enable the communication of the NiFi and MiNiFi instances via NiFi Site-to-Site (STS) protocol, a security via TLS certificates has been enabled among the instances, on a client-server basis. In practical terms, this means that after the deployment of all instances as MARVdash services, some updates of the services configurations are required to copy and make available the adequate certificates pointing to the specific end-points where the NiFis in the different layers have been deployed.

Besides the pure DatAna deployment, although DatAna does not include a message broker as part of the component, docker images of Mosquitto MQTT have been provided to enable the deployment of the message broker at the different layers using MARVdash.

Once deployed, a set of data flows have been defined for each of the inference model output ingestion, transformation and compliance with the data models required by the DFB. These data flows have been prepared in the required DatAna instances and layers in accordance with the use cases implemented in the pilots for M18.

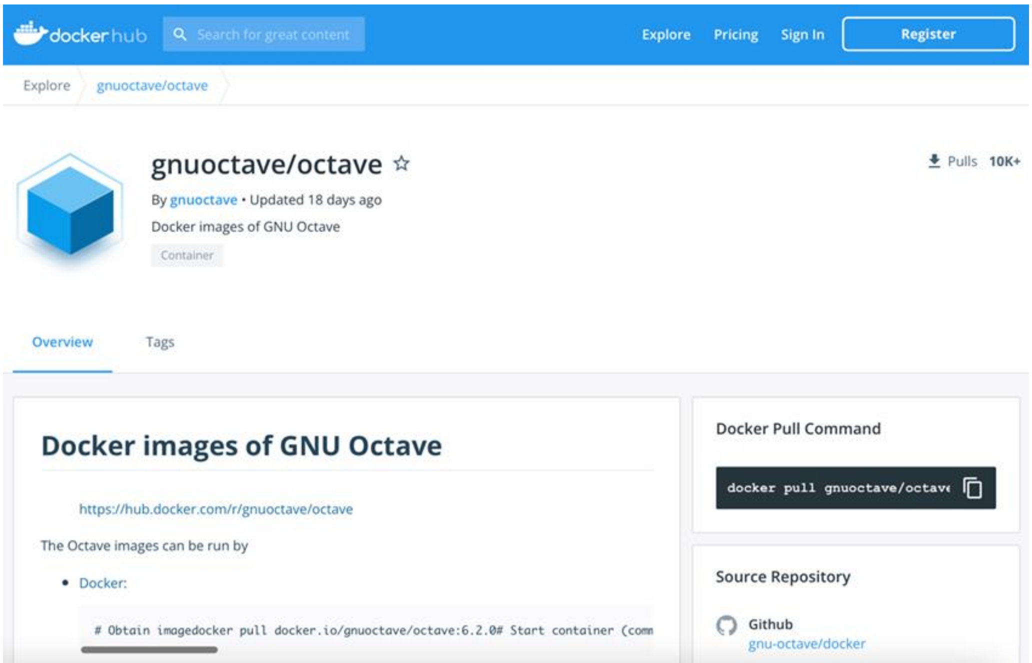
More information about DatAna and its deployment can be found in the MARVEL project deliverable D2.2.

2.3.4 Hierarchical Data Distribution - HDD

Description:

HDD is a set of distributed algorithmic schemes for guaranteeing latency requirements while effectively prolonging network lifetime in wireless edge networks. The current MARVEL design of HDD considers the problem of Apache Kafka data topic partitioning optimisation. Apache Kafka uses partitions to scale a topic across many brokers for producers to write data in parallel, and also to facilitate parallel reading of consumers. Even though Apache Kafka provides some out-of-the-box optimisations, it does not strictly define how each topic shall be efficiently distributed into partitions. The well-formulated fine-tuning that is needed in order to improve an Apache Kafka cluster performance is still an open research problem. HDD first models the Apache Kafka topic partitioning process for a given topic. After that, HDD takes under consideration a set of metrics such as number of brokers, constraints and application requirements on throughput, OS load, replication latency and unavailability, to find how many partitions are needed. This constitutes the formulation of an optimisation problem computationally intractable. Furthermore, HDD implements two simple, yet efficient heuristics to solve the problem: the first tries to minimise and the second to maximise the number of brokers used in the cluster. HDD manages to respect the hard constraints on replication latency and perform efficiently with respect to unavailability time and OS load, using the system resources in a prudent way.

Docker Image:



gnuoctave/octave ☆

By gnuoctave • Updated 18 days ago

Docker images of GNU Octave

Docker images of GNU Octave

https://hub.docker.com/r/gnuoctave/octave

The Octave images can be run by

- Docker:

```
# Obtain imagedocker pull docker.io/gnuoctave/octave:6.2.0# Start container (com
```

Docker Pull Command

```
docker pull gnuoctave/octave
```

Source Repository

Github
gnu-octave/docker

Figure 1: HDD Docker image in Docker Hub

HDD is currently implemented in GNU's Not Unix (GNU) Octave, a high-level programming language primarily intended for scientific computing and numerical computation. Octave helps in solving linear and nonlinear problems numerically, and for performing other numerical experiments using a language that is mostly compatible with MATLAB. The selection of Octave over MATLAB was performed just before we proceeded to the containerisation process of HDD. The commercial licensing of MATLAB greatly influenced the selection process, and Octave was selected in order to avoid complicated licensing constraints in MARVdash.

The docker image of the HDD is therefore dependent on Octave. In order to build the HDD image, we decided to first pull the GNU Octave docker image, as displayed below. The GNU Octave image actually contains an installation on Ubuntu Linux.

As shown in Figure 1, the current version of HDD’s Octave is 6.2.0. We aim at maintaining the versioning of HDD’s Octave updated with respect to the actual Octave Docker image. HDD’s code has then to be uploaded to the running Octave image.

To upload the HDD image on MARVdash, we used the endpoint registry.marvel-platform.eu with our dashboard credentials, as follows (example):

```
docker build –t hdd:2 .
docker image tag gnuoctave/octave:6.2.0 registry.marvel-platform.eu/hddv0:2
docker login registry.marvel-platform.eu
docker push registry.marvel-platform.eu/hdd:2
```

Listing 17: Commands for uploading Docker Image to MARVdash - HDD

After this finished, we were able to view the HDD image in the registry frontend in the dashboard (under "Images").

2.4 E2F2C subsystem

2.4.1 GPURegex

Description:

GPURegex can be deployed to any OpenCL-enabled processor or hardware accelerator, such as discrete GPUs or integrated GPUs. In the first version of GPURegex that has been uploaded to the MARVEL registry and is available to any MARVEL partner, FORTH introduces an implementation for integrated GPUs (i.e., Intel HD Graphics[7]) and an implementation for main processors (i.e., Intel CPUs at URL[8]) for hardware setups that do not offer a GPU. A GPURegex Docker container can be deployed on top of any OpenCL-enabled hardware device. OpenCL drivers are required to be installed in the specific docker container before the execution of GPURegex. Each vendor (e.g., Intel, NVIDIA) and each hardware device (e.g., CPU, discrete GPU, integrated GPU) is supported by vendor and device-specific OpenCL drivers, libraries and runtimes. For instance, the OpenCL drivers that are destined for Intel integrated GPUs are different to those that are destined for NVIDIA GPUs. As already stated, GPURegex is available via two images, uploaded to the MARVEL image registry (i.e., Intel CPU and Intel HD Graphics GPU).

Docker Image:

Once downloaded from the MARVEL docker image registry, the GPURegex component can be deployed by the following certain steps (Listing 18):

```
docker login registry.marvel-platform.eu
docker pull registry.marvel-platform.eu/gpuregex-intel-cpu:1
docker run -it registry.marvel-platform.eu/gpuregex-intel-cpu:1 /bin/sh
```

Listing 18: GPURegex container creation commands

GPURegex returns the input lines that contain patterns that match against them. GPURegex is compiled and executed using the commands (Listing 19):

```
$ make
$ ./bin/gpuregex -p patterns_demo -i input_demo
```

Listing 19: GPURegex execution commands

where -p accepts the pattern file name and -i accepts the input file name. Examples for GPURegex execution are available in Deliverable 4.2.

2.4.2 DynHP

Description:

DynHP is a methodology for training a DNN model and compressing it at the same time. The type of compression operated by DynHP is pruning, i.e., the parameters of a DNN are zero-ed at training time. DynHP operates structured pruning where the idea is to "remove" entire neurons of a DNN or entire convolutional filters. Compression-wise, structured pruning is more

[7] https://marvel-platform.eu/image/gpuregex-intel-gpu

[8] https://marvel-platform.eu/image/gpuregex-intel-cpu

effective since it allows removing entire groups of parameters. DynHP performs hard pruning. With “hard pruning” the parameters that are “switched off” during training cannot be recovered afterwards. DynHP combines structured pruning with hard pruning. Since the structured hard pruning process might degrade the performance of the training, DynHP can adopt a strategy to alleviate that. Precisely, it tunes adaptively the size of the minibatches depending on gradient-related information and the amount of available memory.

Docker Image:

DynHP is developed using the Pytorch Framework and it needs the CUDA11 environment to run on GPUs. Therefore, the DynHP image is based on the `nvidia/cuda:11.1.1-base-ubuntu20.04` image. It is possible to pull an image with CUDA 11.1 using (Listing 20):

```
$ docker pull nvidia/cuda:11.1.1-base-ubuntu20.04
```

Listing 20: Docker pull command

The Docker file of DynHP image is as follows:

```
FROM nvidia/cuda:11.1.1-base-ubuntu20.04

# Remove any third-party apt sources to avoid issues with expiring keys.
RUN rm -f /etc/apt/sources.list.d/*.list

# Install some basic utilities
RUN apt-get update && apt-get install -y \
    curl \
    ca-certificates \
    sudo \
    git \
    bzip2 \
    libx11-6 \
 && rm -rf /var/lib/apt/lists/*

# Create a working directory
RUN mkdir /app
WORKDIR /app

# Create a non-root user and switch to it
RUN adduser --disabled-password --gecos '' --shell /bin/bash user \
 && chown -R user:user /app
RUN echo "user ALL=(ALL) NOPASSWD:ALL" > /etc/sudoers.d/90-user
USER user


# All users can use /home/user as their home directory
ENV HOME=/home/user
RUN chmod 777 /home/user
ENV PATH=/home/user/miniconda/bin:$PATH
```

```
RUN curl -sLo ~/miniconda.sh https://repo.anaconda.com/miniconda/Miniconda3-py39_4.12.0-Linux-x86_64.sh \
 && chmod +x ~/miniconda.sh \
 && ~/miniconda.sh -b -p ~/miniconda \
 && rm ~/miniconda.sh \
 && conda install -y python==3.9 \
 && conda clean -ya

ENV TZ=UTC
RUN sudo ln -snf /usr/share/zoneinfo/$TZ /etc/localtime

RUN mkdir -p /app/dynhp_v1
RUN mkdir -p /app/dynhp_v1/compression-output
RUN mkdir -p /app/dynhp_v1/dataset

COPY .   /app/dynhp_v1/
RUN sudo mv /app/dynhp_v1/home/user/.jupyter /home/user/
RUN sudo chmod 777 -R /app/dynhp_v1

RUN sudo apt-get update  \
    && sudo apt install -y software-properties-common \
    && sudo add-apt-repository ppa:deadsnakes/ppa \
    && sudo apt-get install -y libsndfile1-dev \
    && sudo rm -rf /var/lib/apt/lists/*

WORKDIR /app/dynhp_v1/
RUN python3.9 -m pip install -r requirements.txt
RUN python3.9 -m pip install jupyterlab pandas numpy scipy

# Set the default command to start the jupyter-lab
CMD jupyter-lab --no-browser --port=8686 --ip 0.0.0.0
```

Listing 21: DynHP Dockerfile

The creation of the docker image assumes that the DynHP source code is present on the machine where the image is created. This can be achieved by cloning the DynHP repo as follows:

```
$ git clone git@git.marvel-project.eu:marvel/dynhp/dynhp-compressor.git docker-img
```

Listing 22: Structure of the DynHP root directory

The commands for the Docker image creation are the following:

```
docker build –t dynhp:2 .
docker tag dynhp:2 registry.marvel-platform.eu/dynhp:2
docker login registry.marvel-platform.eu
docker push registry.marvel-platform.eu/dynhp:2
```

Listing 23: Commands for uploading Docker image to MARVdash - dynHP

When executed, the DynHP image returns a shell prompt that can be used to run the compression on the model present in the library and all the necessary python scripts to interact with the AI Model Repository and the MARVEL Data Corpus.

2.4.3 FedL

Description:

FedL is a component developed by UNS for the MARVEL project architecture. FedL contains implementation of high-performance Federated Learning training process for Deep Learning models.

Federated Learning is a training paradigm which allows for distributed privacy-preserving training of Machine Learning (ML) models. In Federated Learning, the training data is never shared with anyone and it is kept at the ingestion source, i.e., partial training is performed near the data source. Typically, in a Federated Learning scenario, there are multiple clients which perform the training with their local data, and a central collection and orchestration point – Federated Learning server, which is then used to perform averaging of all the client models and to provide a global model which is created by combining (averaging) all the client models. This global model is then saved and can be reused for future training or inference. In this way of training, we obtain a model similar to a theoretical model which is trained on all the available client data altogether but, the client data never leaves its source, thus ensuring privacy. Only the parameters (updates, gradients) of the ML models are shared with the server and not the training/input data. If necessary for some models, we can even combine this approach with differential privacy methods by injecting noise into the parameter updates which are shared between the clients and the server. This is another step to ensure that the shared parameter data is harder to reverse engineer to actual training data.

For the MARVEL project needs, FedL component develops a specialised Federated Learning strategy which is meant to optimise the federated learning process in the case of flaky (not consistent) client-server communication. Communication issues are always present in large heterogeneous systems. Addressing these issues is the one of the goals pf this custom federated learning strategy. The Tcustom Non-Uniform Sampling (NUS) strategy (names NUS – non-uniform sampling strategy) allows for some clients to be temporarily unavailable during federated learning. It also saves bandwidth by only requesting client training results if the client data is considered valuable to the global server model, based on several metrics such as number of client data points, model metrics such as accuracy, model gradient variance, client availability history, client training and merging history and others.

Docker Image:

Docker images for the FedL component are a natural extension of the Docker images which are used for Deep Learning models which need to be trained in a Federated Learning process. FedL works by extending the training code of these models to utilise communication between the clients training on local data and the server which is used to perform orchestration.

The FedL server is a model independent component, meaning that its Docker image can be reused for multiple use cases. The configuration of the server (e.g., choice of strategy, number of clients) is configurable on the server side (parameters available also in MARVdash).

We will now briefly show how to build example Docker images for a PyTorch based simple neural network model.

In Listing 24, a snippet from the Dockerfile of FedL server is depicted:

```
FROM python:3.8

# Install flower and dependencies for machine learning
RUN python3 --version
RUN pip3 install torch==1.8.2+cpu torchvision==0.9.2+cpu torchaudio==0.8.2 -f
https://download.pytorch.org/whl/lts/1.8/torch_lts.html
RUN pip3 install flwr==0.17.0

# Copy code in final step so code changes don't invalidate the
# previous docker layers
WORKDIR /opt/marvel
COPY *.py .
COPY *.sh .

EXPOSE 8080/udp
EXPOSE 8080/tcp

# Start the FL server
```

Listing 24: FedL server Dockerfile

For the server image, we need to expose port 8080 (both udp and tcp) for client communication. Code for the custom FedL strategy is imported to the image as well. Llast line runs the FedL server.

A snippet of the Dockerfile for the corresponding client is shown in Listing 25:

```
FROM python:3.8

# Install flower and dependencies for machine learning
RUN python3 --version
RUN pip3 install torch==1.8.2+cpu torchvision==0.9.2+cpu torchaudio==0.8.2 -f
https://download.pytorch.org/whl/lts/1.8/torch_lts.html
RUN pip3 install flwr==0.17.0

# Copy code in final step so code changes don't invalidate the
# previous docker layers
WORKDIR /opt/marvel
COPY *.py .
COPY *.sh .
```

```
# Default values:
ENV CID=1
ENV SERVER_ADDRESS=localhost:8080
ENV NB_CLIENTS=5
ENV EPOCHS=10

# Start the FL client
CMD python3 client.py --cid=$CID --server_address=$SERVER_ADDRESS --
nb_clients=$NB_CLIENTS --epochs=$EPOCHS
```

Listing 25: FedL client Dockerfile

Since we have the same dependencies, layers from the server image can be reused. Here we expose some environment variables such as server address, client id (CID), total number of clients (NB_CLIENTS), number of training epochs per round of Federated Learning training (EPOCHS). These parameters can be externally configured.

Finally, the code snippet in Listing 26 demonstrates how to integrate FedL into existing Deep Learning models. This code is for a TensorFlow/Keras based AVCC model from AU, but it can be adapted to any standard Deep Learning model.

```
class AVCCClient(fl.client.NumPyClient):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.model, self.train_sequence, self.test_sequence, self.val_sequence =
train_backbone(0) # 0 epochs, only to init model

    def get_parameters(self):
        return self.model.get_weights()

    def fit(self, parameters, config):
        self.model.set_weights(parameters)
        self.model = train_backbone(1)
        return self.model.get_weights(), len(self.train_sequence), {}

    def evaluate(self, parameters, config):
        self.model.set_weights(parameters)
        loss, accuracy = self.model.evaluate(self.test_sequence)
        return loss, len(self.test_sequence), {"accuracy": accuracy}
```

Listing 26: Integration of FedL into existing Deep Learning models

As it can be seen from the code, FedL client is completely model agnostic, we just need the interfaces to train the model, and to obtain its parameters and metrics as is.

2.5 System outputs subsystem

2.5.1 SmartViz

Description:

The system outputs of MARVEL are realised through the Decision-Making Toolkit (DMT) which aims at assisting stakeholders in short and long-term decision-making. DMT is based on Zelus' SmartViz component which is a collection of advanced visualisation tools, offering multi-purpose data representations and visualisations.

SmartViz is a versatile data visualisation solution that empowers domain experts to discover patterns, behaviours, and correlations of data items. It consists of a set of visualisation tools developed to allow a more straightforward exploratory analysis of data by using interactive presentations, intuitive monitoring dashboards, and configurable visual representations. SmartViz can visualise data coming in real-time or in batch mode and it can be used to provide visualisation configurations, covering the needs of a variety of users that are required to address all MARVEL stakeholders.

Using its Data Intake adapters, SmartViz is capable of connecting with multiple data sources and then uses its internal data API and configuration options to produce predefined as well as user-defined visualisation dashboards. The output of the adapters is handled by a middleware, that transforms information into internal data representations, which can afterwards feed the visualisations. The Frontend part of the tool is served as a web application directly accessible by end-users.

Docker Image:

SmartViz consists of two images, one for the Frontend and one for the Middleware part, and it also uses a nginx web server to act as proxy for the internal services. There are two Dockerfiles as seen below for each part of the application and a docker-compose.yaml file.

```
Middleware Dockerfile:
FROM node:14
WORKDIR /usr/src/app/srv
COPY package.json package-lock.json .
RUN npm install
COPY . .
CMD [ "node", "server.js" ]

Frontend Dockerfile:
FROM node:14 As builder
WORKDIR /usr/src/app/smartviz
COPY package.json package-lock.json ./
RUN npm install
COPY . .
RUN npm run build -- --base-href='/smartviz/'
FROM nginx
COPY --from=builder /usr/src/app/smartviz/dist /usr/share/nginx/html/smartviz
#COPY ./nginx.conf /etc/nginx/conf.d/default.conf
```

```
CMD ["/bin/sh",  "-c",  "envsubst <
/usr/share/nginx/html/smartviz/assets/env.template.js >
/usr/share/nginx/html/smartviz/assets/env.js && exec nginx -g 'daemon off;'"]

Docker-compose.yaml:
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  annotations:
    nginx.ingress.kubernetes.io/force-ssl-redirect: '"true"'
    nginx.ingress.kubernetes.io/proxy-read-timeout: "3600"
    nginx.ingress.kubernetes.io/proxy-send-timeout: "3600"
    nginx.ingress.kubernetes.io/proxy-body-size: "0"
  name: $NAME
spec:
  rules:
  - host: $HOSTNAME
    http:
      paths:
        - backend:
            serviceName: $NAME
            servicePort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: $NAME
spec:
  type: ClusterIP
  ports:
  - port: 80
  selector:
    app: $NAME
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-config
data:
  default.conf: |
    upstream server {
        server 127.0.0.1:8000;
    }
    upstream smartviz {
        server 127.0.0.1:8080;
    }
    server {
        listen 80;
        location /smartviz {
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection "upgrade";
            proxy_pass http://smartviz;
        }
        location /server {
            rewrite ^/server/(.*) /$1 break; # works for both /server and
/server/
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection "upgrade";
            proxy_pass http://server/;
        }
    }
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: smartviz-config
data:
  default.conf: |
    #refresh page -> point index to get the route info
    server {
      listen 8080;

      location / {
        root /usr/share/nginx/html; #nginx root html
        index index.html index.htm;
        try_files $uri $uri/ /smartviz/index.html =404; #subfolder index path
      }
      include /etc/nginx/extra-conf.d/*.conf;
    }
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: $NAME
spec:
  replicas: 1
  selector:
    matchLabels:
      app: $NAME
  template:
    metadata:
      labels:
        app: $NAME
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
        volumeMounts:
        - name: nginx-config-volume
          mountPath: /etc/nginx/conf.d/default.conf
          subPath: default.conf
      - name: server
        image: stellamarkop/dmtserv:$VERSION
        ports:
        - containerPort: 8000
        env:
        - name: PORT
          value: "8000" #server running internally at 8000; if you want to
change the port you should change the default.conf && exposed Port accordingly.
        - name: KAFKA_URL
          value: $KAFKA #set kafka IP + Port (broker)
        - name: TOPIC
          value: $TOPIC #topic env var format MUST be: topic1
        - name: EL
          value: http://$ELASTICSEARCH #set elastic search IP + Port
        - name: INDEX
          value: $INDEX #set one available index of elastic
      - name: smartviz
        image: stellamarkop/dmtsmartviz:$VERSION
        ports:
        - containerPort: 8080
        env:
        - name: SERV_HOST
          value: https://$HOSTNAME/server
        - name: SOCKET_HOST
          value: https://$HOSTNAME #server base IP for Socket.io; adjust the IP
url according your host environment. Server runs under nginx proxy pass(thats
why port is 4200)- we have configured in Angular to request Socket under /server
subdomain.
        volumeMounts:
        - name: smartviz-config-volume
          mountPath: /etc/nginx/conf.d/default.conf
          subPath: default.conf
      volumes:
      - name: nginx-config-volume
        configMap:
          name: nginx-config
          defaultMode: 0644
      - name: smartviz-config-volume
        configMap:
          name: smartviz-config
          defaultMode: 0644
---
kind: Template
name: SmartViz
description: SmartViz frontend
singleton: yes
datasets: no
variables:
- name: NAME
  default: smartviz
- name: HOSTNAME
  default: smartviz.example.com
- name: VERSION
  default: test
  help: Container version/tag
- name: KAFKA
  default: kafka:9092
  help: Kafka service endpoint
- name: TOPIC
  default: test
  help: Kafka topic
- name: ELASTICSEARCH
  default: example:9200
  help: Elasticsearch service endpoint
- name: INDEX
  default: mappings
  help: Elasticsearch index
```

Listing 27: SmartViz Dockerfile

2.5.2 MARVEL Data Corpus-as-a-Service

Description:

The MARVEL Data Corpus is going to store complete datasets (with anonymised and annotated data). It receives data from the piloting environments (e.g., video/audio for surveillance cameras) and stores it in a Big Data repository. Then, the user, either internal (other MARVEL components and partners) or external (research and industrial communities), can search and download the underlying files and facilitate machine learning (ML) processes.

The core file repository is implemented by the Hadoop Distributed Files System (HDFS), while the management of this Big Data database is performed via HBase. There are also components that offer the interface between the administrator user and these elements, such as the Ambari web interface and the ELK stack. The HBase/Hadoop system is comprised of several subcomponents. The data files themselves are stored in *HDFS Data Nodes*. In a clustered environment, that data storage is distributed among several *Region Servers*, with every Region Server controlling a set of Data Nodes. The *HBase Master* manages the assignment of these regions and the main database operations (e.g., create, update, delete tables, etc.). Then, *ZooKeeper*, which is part of HDFS, retains a live cluster status. Finally, the *Name Node* maintains metadata information for all physical data blocks. A high-level abstraction is depicted in the following figure.

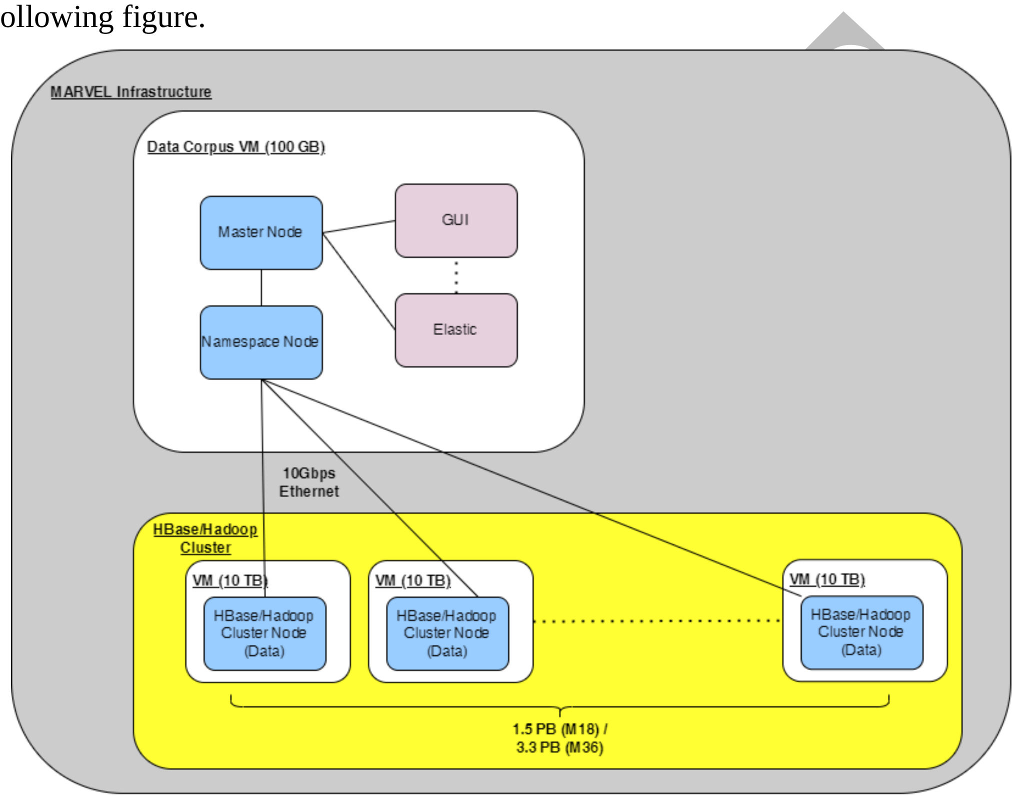


Figure 2: Data Corpus Infrastructure

The Corpus also deploys augmentation techniques. The user can apply them in order to create augmented versions of the existing datasets. For example, the user can adjust the brightness of a video that was recorded during morning time to represent the same results in the afternoon time. Then, the ML components can also parse this data and create a more robust evaluation process for the case where they will have to process a livestream during afternoon.

Moreover, there are JAVA applications that implement the programmable interfaces with the repository and develop functionality, such as storing a file or whole dataset, updating existing files/datasets, search for information and retrieving the related files, etc.

Also, there are graphical interfaces that assist the use of the Corpus by the end-user. The user can review the already ingested datasets and their content, as well as update them or upload new ones.

Concerning the deployment of the system, all these components are dockerised. In the MARVEL backend/cloud server and the MARVdash, there is the main Data Corpus VM. This includes the elements of the Master HBase/Hadoop Node, the Name Node, the graphical interfaces, the Python augmentation libraries, and the JAVA applications that implement the programmable interfaces and the integration with other MARVEL components (i.e., DFP and StreamHandler).

Then, for the cluster, there is one VM for each Data Node. Each VM has a maximum hard disk space (e.g., 300TB), which is managed by the Data Node. As the volume of the ingested datasets is increased, VMs are added. The goal is to reach 3.3 PB storage until M36.

Docker Image:

A set of Docker images have been created and deployed for Data Corpus. The following list summarises them:

Hadoop:

- hadoop-namenode:2.0.0-hadoop2.7.4-java8
- hadoop-datanode:2.0.0-hadoop2.7.4-java8
- hadoop-nodemanager:2.0.0-hadoop2.7.4-java8
- hadoop-resourcemanager:2.0.0-hadoop2.7.4-java8
- hadoop-historyserver:2.0.0-hadoop2.7.4-java8

HBase:

- hbase-master:1.0.0-hbase1.2.6
- hbase-regionserver:1.0.0-hbase1.2.6
- Zookeeper
- zookeeper:3.4.10
- Ambari
- docker-ambari

ELK:

- elasticsearch:elastdocker-7-17.0
- logstash:elastdocker-7-17.0
- kibana:elastdocker-7.17.0

Python augmentations:

- augmentation_libraries

JAVA application:

- docker-hbase_fileservice

GUI:

- aungular-gui_service

3 E2F2C deployment approach

This section is devoted to the description of the main elements of the MARVEL E2F2C framework. In the first subsection, we describe i) what is Kubernetes; ii) cluster architecture; iii) containerisation technique that is used; and iv) the necessity of a VPN. The second subsection depicts how a suitable execution environment is chosen for each MARVEL component deployment, and the third section talks about the next steps.

3.1 Architecture of the MARVEL E2F2C framework

3.1.1 Kubernetes

Kubernetes is a portable, extensible, open-source platform for managing containerised workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. The name Kubernetes originates from Greek language, meaning helmsman or pilot. K8s as an abbreviation and results from counting the eight letters between the "K" and the "s". Google open-sourced the Kubernetes project in 2014 [1].

The main advantages of Kubernetes are:

- It manages containers. Containers are lightweight, portable and immutable.
- It is distributed in nature (=Kubernetes cluster). It takes care of network, resource management, scaling, and resource failures.
- It runs “everywhere” (any scale, any architecture). Any host machine with Docker and Kubernetes tools installed can be part of a Kubernetes cluster. That makes Kubernetes OS independent and hardware independent.

Kubernetes is an orchestration engine for container technologies. Kubernetes can make the deployment process faster and easier and also run updates with almost zero downtime. Furthermore, Kubernetes can detect and restart services when a process inside a container crashes. One of the advantages of the container orchestration is that the user traffic is load balanced across the various containers. In case of running out of hardware resources, if the nodes are scaled appropriately applications don’t fail. Furthermore, Kubernetes allows you to mount and add storage to run stateful applications.

The most basic objects in Kubernetes are:

- Pod, which is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers.
- Deployment, which is an object that is constituted by a collection of pods defined by a template and a replica count. Replica count indicates the number of pods/containers we want to run.
- Service, which is an object that provides a stable endpoint in order to direct traffic to the desired pods. This endpoint remains the same even if the aforementioned pods change.
- Ingress, which is the object which exposes an endpoint of our application to traffic external to Kubernetes cluster, typically Hypertext Transfer Protocol (HTTP).

3.1.1.1 Cluster Architecture

When Kubernetes tools are deployed, a cluster is initiated. A Kubernetes cluster, depicted in Figure 3, consists of a set of worker machines, which in Kubernetes are called nodes. Nodes run containerised applications. Every cluster has at least one worker node. The worker node(s) host the Pods which are the components of the application workload. The control plane manages the worker nodes and the Pods in the cluster.

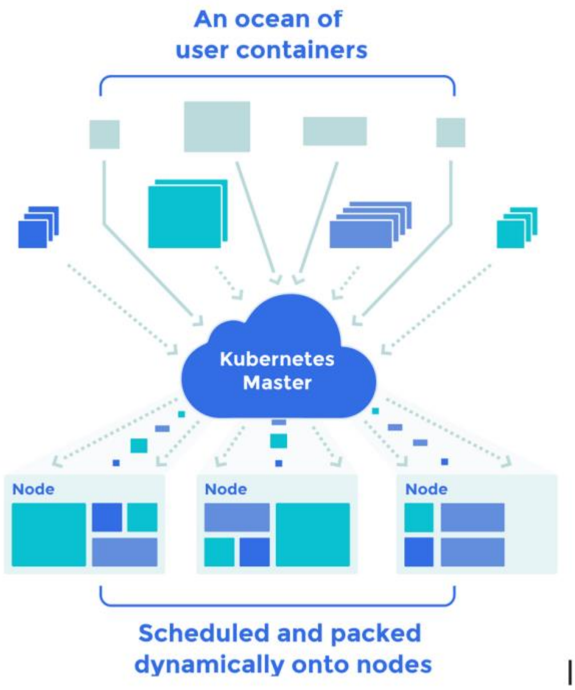

An ocean of user containers
Kubernetes Master
Node
Node
Node
Scheduled and packed dynamically onto nodes

Figure 3: Kubernetes cluster architecture[9]

In the MARVEL environment, the control plane is installed on a VM deployed on PSNC's OpenStack offering. Other VMs deployed on PSNC will also become nodes of the Kubernetes cluster and will form the Cloud of the MARVEL architecture. The devices that form Edge and Fog will become nodes of this Kubernetes Cluster, respectively.

The control plane's components are responsible for making global decisions. This includes scheduling, as well as detecting and responding to cluster events. If, for example, a deployment's replicas fails, then a new pod will start.

[9] source: https://discuss.newrelic.com/t/relic-solution-what-you-need-to-know-about-new-relic-when-deploying-with-docker/52492

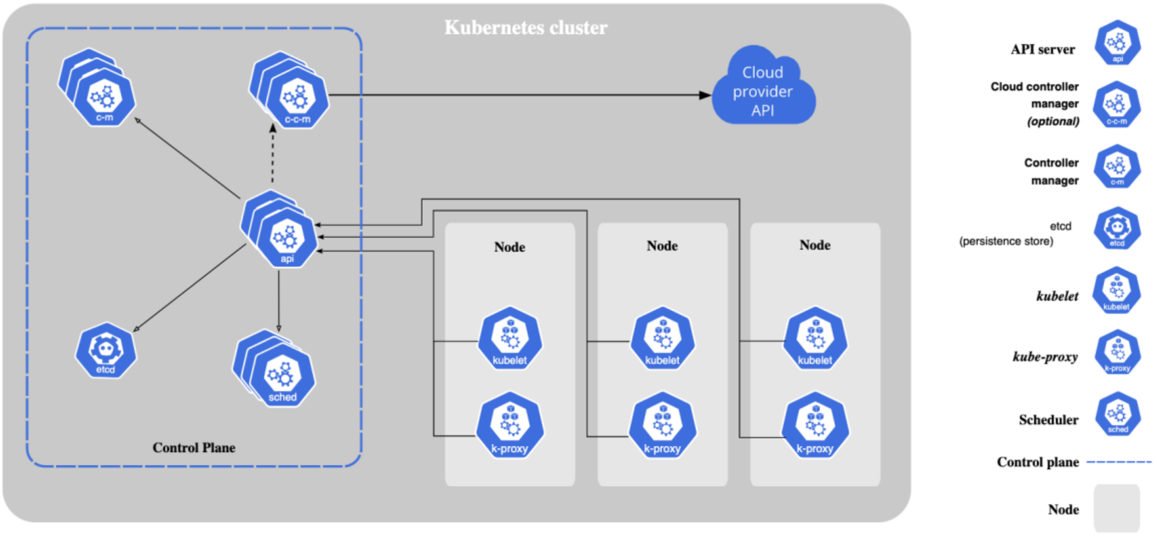


Figure 4: Kubernetes main components[10]

The main components of the control plane, included in Figure 4, are:

- kube-apiserver, which is a component that exposes the Kubernetes API.
- etcd, which is a consistent and highly-available key-value store used as Kubernetes' backing store for all cluster data.
- kube-scheduler, which is the control plane component that watches for newly created Pods with no assigned node, and selects a node for them to run on.
- kube-controller-manage, which is the control plane component that runs controller processes.
- cloud-controller-manager, which is the Kubernetes control plane component that embeds cloud-specific control logic.

Node components run on every node on each layer, maintaining running pods and providing the Kubernetes runtime environment. These components are:

- kubelet, which is an agent that runs on each node in the cluster and is responsible for reassuring that containers are running in a Pod.
- kube-proxy, which is a network proxy that also runs on each node of the cluster, implementing part of the Kubernetes Service concept.
- Container runtime, which is the software that is responsible for running containers.

3.1.1.2 Containers

Docker containers have become the de-facto standard format and are well adopted by the community of the software developers. Containerisation, in general, is categorised as a virtualisation technology in a lightweight form. A container is able to package an application along with its dependencies and its execution environment into a unit used for software development and running of an application on any system [2]. A Dockerfile is the mean to define the contents of a Docker container in a declarative way, including instructions for software deployment, variable definition, command execution, etc. [3], following the concept

[10] source: https://kubernetes.io/docs/concepts/overview/components/

of Infrastructure-as-Code (IaC) [4]. IaC allows for easier distribution and edit of configurations, and provisioning of the same environment every time.

A virtualisation technique at the operating system level is used by containers to achieve process and network isolation. Linux Containers (LXC) could be considered as forerunner achieving isolation with the use of chroot, cgroups, and namespaces. Docker containers extend LXC offering additional functionality, such as portable container images. A docker container image is an object that includes the contents of a container and can be easily transferred and deployed across individual hosts. Containers are considered least resource intense than the other virtualisation techniques, like Virtual Machines (VMs). In comparison with a VM, containers use the kernel of their hosts and do not emulate a whole operating system, reducing the necessary resources [5].

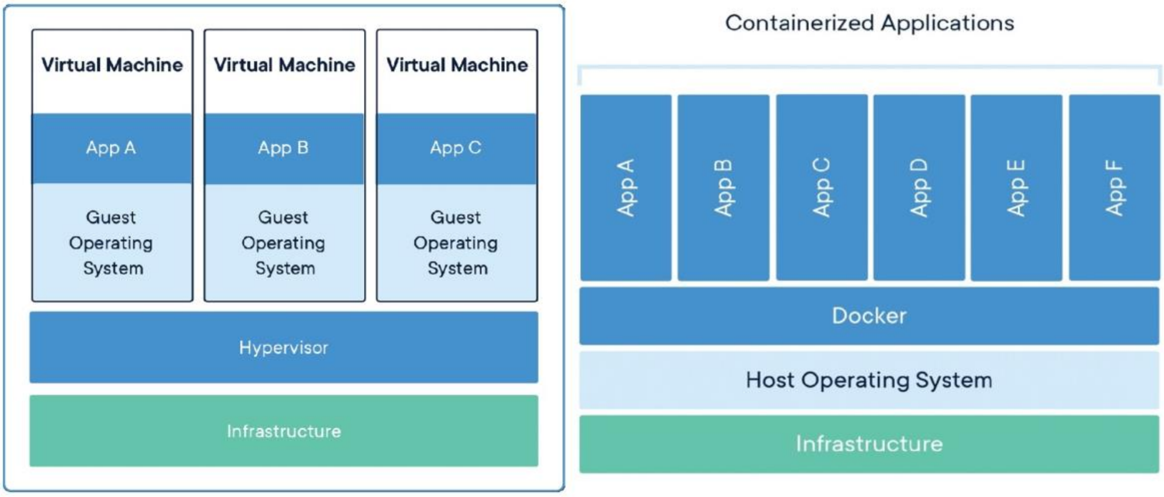


Figure 5: Docker basic concepts[11]

The Dockerfile is a text document that includes declarative instructions that describe the contents of a Docker container. In simpler words, a Dockerfile contains all the commands that can be called on the command line to build a Docker image. Listing 28 depicts such a Dockerfile including some of the most used instructions. What follows is a short explanation of the included instructions to offer a basic understanding of the Dockerfile format. The FROM instruction specifies the base image on which the corresponding Docker container will be built. It can be an operating system or another existing container. According to best practices, this initial image should be as minimal as possible. The FROM instruction is mandatory for every Dockerfile. In this example, the base image is the one of ubuntu operating system, version 18.04.

An optional instruction is that of the MAINTAINER. This instruction refers to the name and email of the maintainer of the Dockerfile. Setting environmental variables with an instruction is also an option. The ENV instruction initialises a variable. In our example, it is used for the definition of a Kafka instance endpoint (IP address and port) and a corresponding topic.

RUN is a rather general instruction that allows for the execution of any shell command within the container. Most of the time, it is used to retrieve dependencies, compile and install software. In our example, it is used for java and maven installations.

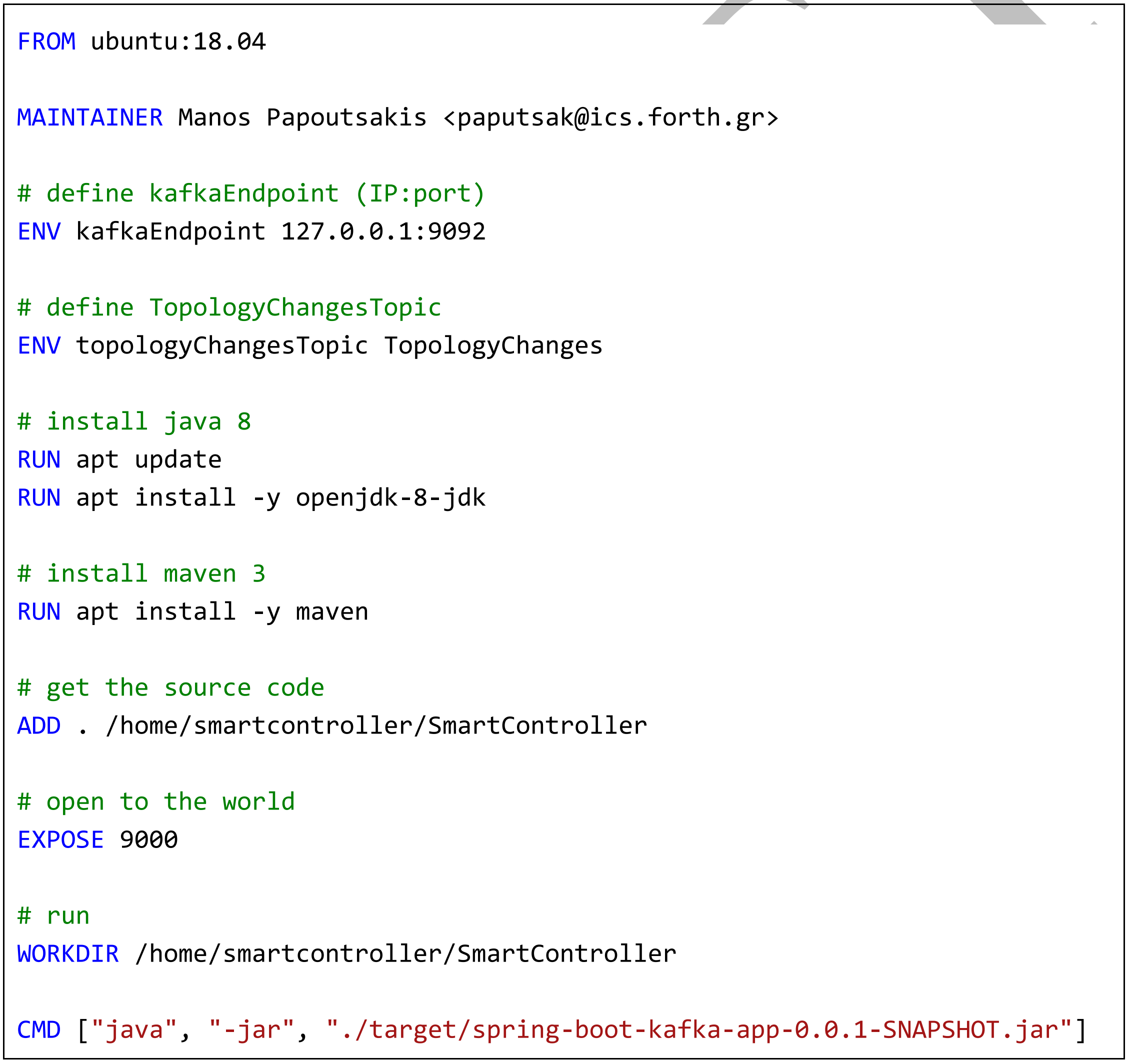
[11] source: https://borosan.gitbook.io/docker-handbook/basic-consepts

ADD and its sister instruction COPY are used for transferring files into the newly created container from the host. ADD, on top of the transfer functionality, can use a URL as source path and unpack compressed files.

The EXPOSE instruction specifies a network port for the Docker container to enable network communication. This port is the port that the underlying container process listens to. Port 9000 is exposed in our example.

WORKDIR instruction specifies the working directory of the container. Other instructions such as RUN, CMD, ADD, COPY will be executed in this defined directory. We could say that WORKDIR includes mkdir and cd commands.

Finally, CMD instruction represents the command that the container executes when the built image is launched and the container is started. If more than one CMD instructions are included in a Dockerfile, only the last one is executed. The difference with the RUN instruction is that the latter creates a new intermediate image layer on top of the previous one. In our example, the CMD instruction runs a jar file that will start the process of the container.



```
FROM ubuntu:18.04

MAINTAINER Manos Papoutsakis <paputsak@ics.forth.gr>

# define kafkaEndpoint (IP:port)
ENV kafkaEndpoint 127.0.0.1:9092

# define TopologyChangesTopic
ENV topologyChangesTopic TopologyChanges

# install java 8
RUN apt update
RUN apt install -y openjdk-8-jdk

# install maven 3
RUN apt install -y maven

# get the source code
ADD . /home/smartcontroller/SmartController

# open to the world
EXPOSE 9000

# run
WORKDIR /home/smartcontroller/SmartController

CMD ["java", "-jar", "./target/spring-boot-kafka-app-0.0.1-SNAPSHOT.jar"]
```

Listing 28: Dockerfile snippet

3.1.1.3 Load balancing and networking

Kubernetes has its own network model according to which each Pod has its own IP address. Pods are considered instances of running processes in the Kubernetes cluster. There is no need for creating links between Pods and mapping container ports to host ports. As a result, as far as network functionalities, such as port allocation, naming, service discovery, load balancing, application configuration, and migration, are concerned, Pods are treated as VMs or normal physical hosts, i.e., part of a network.

The requirements of a Kubernetes implementation are the following:

- A pod can communicate with all pods in different nodes without NAT.
- Agents on a node (e.g., system daemons, kubelet) can communicate with all pods of the node.
- Pods pin the host network of a node can communicate with all other pods of other nodes without NAT (applies to Linux).

Moreover, IP addresses of a Kubernetes network make sense only at the Pod scope. Containers included in a Pod have their own network namespace (IP and MAC address). According to that, containers of the same pod reach each other using localhost and have to coordinate regarding the ports they are using. As a result, containers within a Pod communicate with each other via loopback, while Pods themselves use the cluster networking for the communication at their level.

Regarding the exposure of an application to the outside of the Kubernetes cluster world, a Kubernetes Service can be used. A running application may use a set of Pods. Kubernetes gives to those Pods an IP address and a DNS name, while it undertakes the load-balancing across them. A Service defines a logical set of Pods and a policy by which to access them. The set of Pods targeted by a Service is usually determined by a selector. The existence of a Service is justified by the non-permanent nature of the Pods. They can be created and destroyed depending on the current state of the cluster and desired application deployment. Services hide the complexity of keeping track of the set of Pods that an application is coupled with.

Listing 29 depicts a Service object and how it is defined in Kubernetes. This piece of code creates a Service with name “nginx”. This Service targets TCP port 80 on any Pod with label “app.kubernetes.io/name: webserver”, as the *selector* defines. Moreover, the *targetport webserver-service* of the created Service is bind to port 80 of the Pod. The *webserver-service* is the name that is given to a specific port, defined in the corresponding Pod object.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  selector:
    app.kubernetes.io/name: webserver
  ports:
  - name: port-name
    protocol: TCP
    port: 80

    targetPort: webserver-service
```

Listing 29: Service object snippet

Finally, the Service object can be used for exposing application functionality only to the internal of a Kubernetes cluster.

3.1.2 Virtual Private Network – VPN

3.1.2.1 VPN architecture

The VPN solution that is used in the MARVEL E2F2C framework is based on the n2n[12] architecture. As it can be seen in Figure 6, there are two key components: Edge Nodes and Super Nodes. The Edge Nodes are the peers participating in the network, while the Super Nodes are used by the Edge Nodes for discovering other Edge Nodes. Moreover, Super Nodes are used for routing the traffic when the nodes are behind symmetrical firewalls.

Due to the presented architecture, a peer-to-peer network is created that works on the second layer of the OSI model[13], allowing the peers to cross NAT and firewalls and being reachable. Edge nNodes that participate in the same virtual network form a community. Super Nodes are able to serve more than one community and a single computer can join multiple communities. Within a community encryption of the packets is feasible with the use of an encryption key. Edge nNodes establish direct communication among themselves via UDP, but when this is not possible, due to special NAT circumstances, then the Super Node can facilitate the relay of the packets.

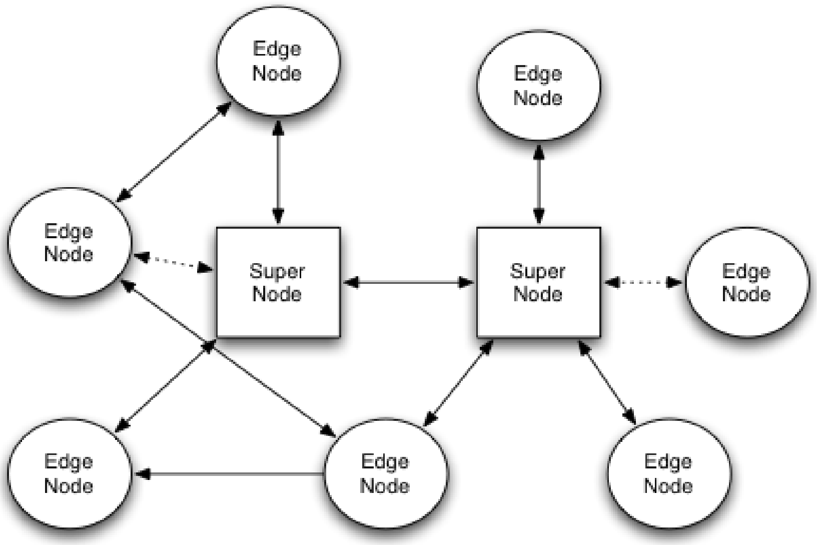


Figure 6: n2n VPN architecture - source: https://www.ntop.org/products/n2n

[12] https://www.ntop.org/products/n2n/

[13] https://www.imperva.com/learn/application-security/osi-model/

3.1.2.2 Necessity of VPN

As it is already mentioned, Kubernetes is the backbone of the MARVEL E2F2C framework, where most of the MARVEL components will be deployed. However, many components are not going to be deployed within the actual network that the original Kubernetes nodes have created (cloud layer). This raised the need for nodes that exist at the fog and at the edge layer to join the Kubernetes cluster at the cloud.

Kubernetes by design requires that all pods can communicate with other pods on any node without NAT which comes in direct contradiction with the actual setup of having remote nodes. VPN provides the solution here, due to the fact that it brings together all the participating nodes as if they were under the same local network making any NAT or firewall transparent to the communication between them (see Figure 7). VPN becomes the undelaying network that allows each remote node to join the Kubernetes cluster at the cloud. This implies that all components that are deployed in Kubernetes traverse the tunnel created by the VPN.

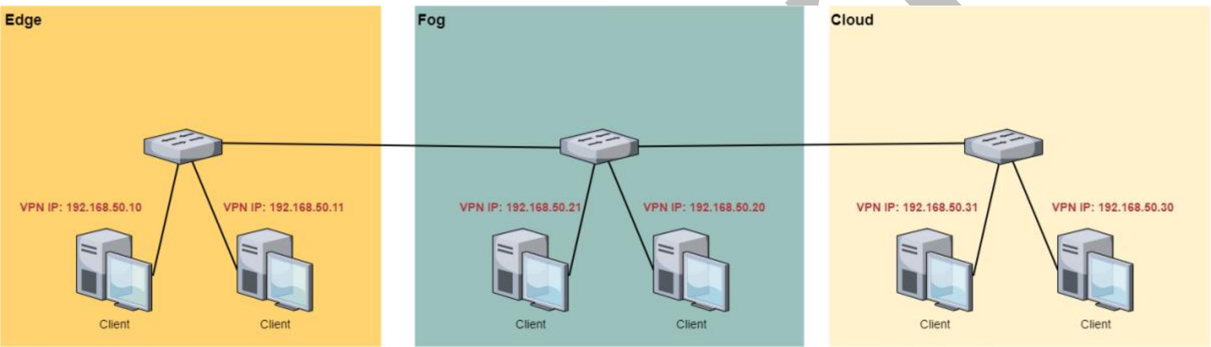


Figure 7: VPN creates an **underlaying** network for remote Kubernetes nodes

3.1.2.3 Implementation

The implementation of VPN in one of the project use cases that GRN is leading, is described in this section. The infrastructure of GRN consists of a server that is located at the fog layer and a workstation located at the edge layer (see Figure 8). A Super Node that assigns a VPN IP for both the server and the workstation is used in the cloud in PSNC's infrastructure. In that way, the two machines in GRN join the same network with MARVdash and become part of the existing Kubernetes network. Nodes are able to directly announce themselves and discover other nodes via the Super Node.

The communication between the participating nodes is limited to the traffic that matches the network subnet defined by the EdgeSec VPN. This means that all unrelated traffic such as browsing the internet or downloading updates is not routing through the VPN thus limiting the overhead of the VPN channel.

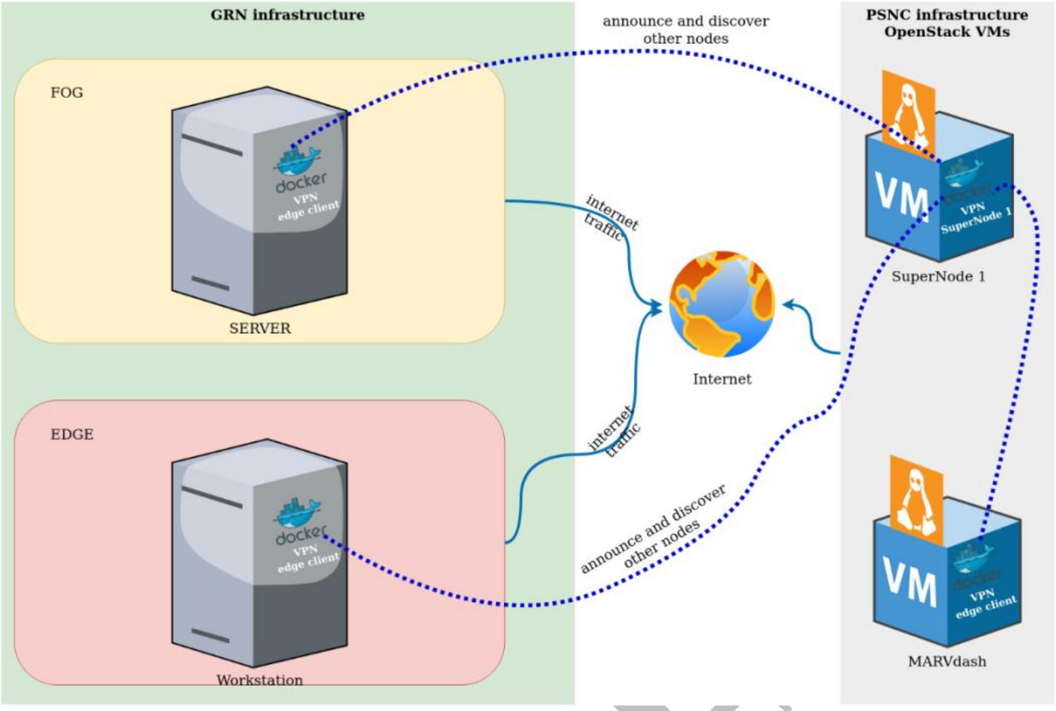


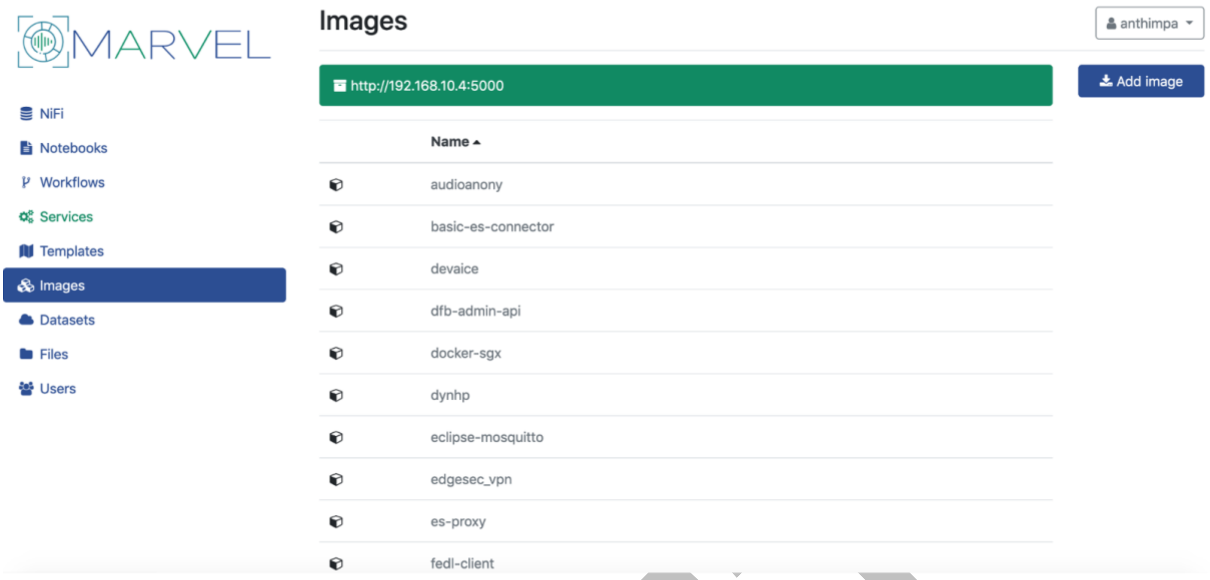
Figure 8: VPN implementation in GRN use cases

3.2 Deployment method

The deployment method though MARVdash is described in the following steps:

1. **Upload the component docker image to MARVdash**. To upload your image, use the endpoint registry.marvel-platform.eu with your dashboard credentials. For example, to upload a container image called "myimage", use the following commands:
   - docker build -t myimage:2
   - docker tag myimage:2 registry.marvel-platform.eu/myimage:2
   - docker login registry.marvel-platform.eu # Only required once
   - docker push registry.marvel-platform.eu/myimage:2

   After this finishes, you should be able to view your image in the registry frontend in the dashboard (under "Images"), as depicted in Figure 9.



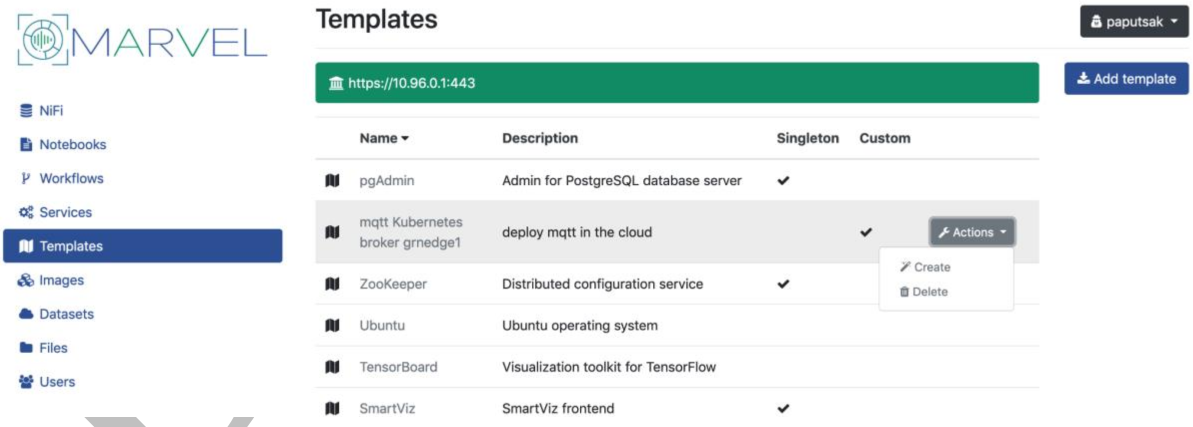
Images

http://192.168.10.4:5000

Name ▴
audioanony
basic-es-connector
devaice
dfb-admin-api
docker-sgx
dynhp
eclipse-mosquitto
edgesec_vpn
es-proxy
fedl-client

Figure 9: Docker images tab in MARVdash UI

2. **Create the corresponding YAML file (Template)**. MARVdash provides a way for users to easily configure and start services, by integrating a service templating mechanism based on Helm. This YAML file should include information such as available APIs of the service, description of the node that the service is deployed at, container image, etc. The Template YAML must be uploaded from the menu Templates and clicking on the button "Add template" (see Figure 10).



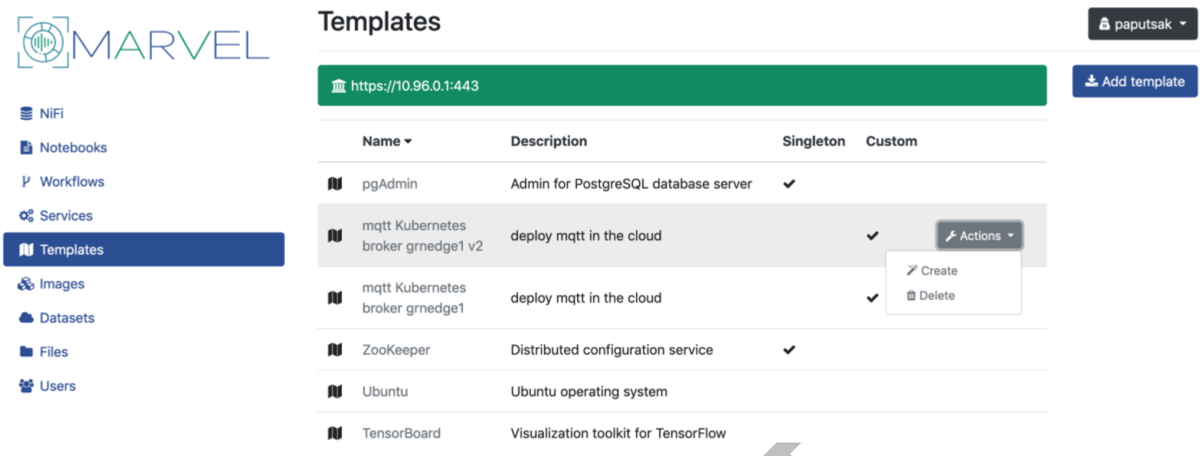
Templates

https://10.96.0.1:443

Name ▾	Description	Singleton	Custom
pgAdmin	Admin for PostgreSQL database server	✓	
mqtt Kubernetes broker grnedge1	deploy mqtt in the cloud		✓
ZooKeeper	Distributed configuration service	✓	
Ubuntu	Ubuntu operating system		
TensorBoard	Visualization toolkit for TensorFlow		
SmartViz	SmartViz frontend	✓	

Figure 10: Templates tab in MARVdash UI

3. **Deploy the service based on the YAML file of step 2**. This step is just a click on the Actions button of the corresponding Template on the Template page of MARVdash (see Figure 11).



Name	Description	Singleton	Custom
pgAdmin	Admin for PostgreSQL database server	✓	
mqtt Kubernetes broker grnedge1 v2	deploy mqtt in the cloud		✓
mqtt Kubernetes broker grnedge1	deploy mqtt in the cloud		✓
ZooKeeper	Distributed configuration service	✓	
Ubuntu	Ubuntu operating system		
TensorBoard	Visualization toolkit for TensorFlow		

Figure 11: Deployment of a service though MARVdash UI

3.1.3 Taints/Tolerations

3.1.3.1 Concepts

One of the mechanisms Kubernetes offers are node taints and tolerations. These are similar to applying node affinity rules but from a different perspective. So, while affinity rules have as a target goal to attract pods to specific nodes, a tainted node repels a set of pods. In order for any pod to run on these nodes, tolerations have to be applied to them.

Tolerations are applied to pods, and allow (but do not require) the pods to schedule onto nodes with matching taints. Taints and tolerations cooperate and ensure that pods are scheduled to appropriate nodes.

A tainted node can have three possible effects from the perspective of Kubernetes scheduler:

- NoSchedule, which leads the Kubernetes scheduler to only schedule pods that have the right tolerations for the tainted nodes.
- PreferNoSchedule, which leads the Kubernetes scheduler to try to avoid scheduling pods that don’t have the right tolerations for the tainted nodes.
- NoExecute, which leads Kubernetes tools to evict the running pods from the nodes if the pods don’t have the right tolerations for the tainted nodes.

3.1.3.2 Concept usage

A pod is the smallest deployable unit of computing that you can create and manage in Kubernetes. When a pod is created in Kubernetes, the scheduler is trying to assign it to each node. If no restrictions and limitations are applied, the scheduler places the pods across the nodes to balance them equally.

Taints and tolerations are mainly used for cases where the nodes have to be dedicated for a reason. One of these reasons could be the necessity of a dedicated set of nodes for exclusive use. For example, this set of nodes could be used by a particular set of use cases. In order for this to be achieved, the node must be tainted and then the pods to have tolerations. Only the pods with the tolerations will have access to the tainted nodes, along with any other nodes in the cluster. Another motivation for tainting nodes is if they have special hardware. If for example, a small subset of nodes has GPUs then it would be desirable to keep pods that don't need the specialised hardware outside of those nodes.

As mentioned above for each use case, we have particular resources. So, we want only pods from these use cases to run on these nodes. By default, pods have no tolerations. So, when we taint a node, no unwanted pod can access it. In order to achieve the desired pods to access the tainted node, we add toleration to these pods. Taints are set on nodes and tolerations are set to pods.

Taints “tell” the node to only accept pods with certain tolerations.

3.1.3.3 MARVEL examples

For the MARVEL Kubernetes cluster to better serve the MARVEL use cases, we taint nodes of each layer (Fog, Edge) in order to properly assign pods. In the example of the GRN Edge host machine after the installation of the Kubernetes tools and the EdgeSec VPN, each host machine in each layer is part of the MARVEL Kubernetes cluster.

So, in the case of the GRN Edge host machine, the node is tainted with the following command (Listing 30)

```
kubectl taint nodes grnedge1 Layer=GRNEDGE1:NoSchedule
```

Listing 30: Taints example for the GRN Edge host machine

which means that Kubernetes Scheduler can assign to this node only pods that have toleration added to them. So, in order for the pods to be assigned to this node, the following lines have to be added to the corresponding YAML file. (Listing 31).

```
tolerations:
      - key: "Layer"
        operator: "Equal"
        value: "GRNEDGE1"
```

Listing 31: Toleration example for the GRN Edge host machine

Kubernetes scheduler will only schedule pods that have the right tolerations (Layer Equal to GRNEDGE1) for the tainted node.

3.1.4 Affinity

3.1.4.1 Concept

Node affinity is a set of rules, which when applied help the scheduler to decide on which node of the cluster to place the pod. This decision is taken with the use of selectors. A Kubernetes Selector allows for the selection of Kubernetes resources based on the value of labels and resource fields assigned to a group of pods or nodes.

In order for the admin of the Kubernetes cluster to set the aforementioned rules, the nodes have to be labelled and in each pod’s definition label selectors have to be defined. Node affinity allows a pod to specify an affinity towards a group of nodes, so it can preferably be scheduled on them. The simplest way the admin can add the node selection constraint is by using the *nodeSelector*. So, in order for the pod to run on the node, it should have defined the labels. Node affinity has a similar approach and allows the administrator of the Kubernetes cluster to limit the nodes where pods can be executed.

The affinity feature consists of two types of affinity. The first is Node affinity functions, which are like the *nodeSelector* field but are more expressive and allow you to specify soft rules. The other is inter-pod affinity/anti-affinity, which allows you to attract or repel Pods.

Node affinity may block or not the scheduling of a pod as described in the two cases below.

- requiredDuringSchedulingIgnoredDuringExecution: The scheduler can't schedule the Pod unless the rule is met.
- preferredDuringSchedulingIgnoredDuringExecution: The scheduler tries to find a node that meets the rule. If a matching node is not available, the scheduler still schedules the Pod.

If you want to dedicate the nodes for pods and ensure they only use the dedicated nodes, then you should additionally add a label similar to the taint to the same set of nodes (e.g., dedicated=usecase), and the admission controller would additionally add a node affinity.

3.1.4.2 Concept usage

Node affinity is a property of Pods that attracts them to a set of nodes either as a preference or a hard requirement. Tolerations are applied to pods, and allow, but do not require, the pods to schedule onto nodes with matching taints. Taints are the opposite since they allow a node to repel a set of pods. Taints and tolerations work together to ensure that pods are not scheduled onto inappropriate nodes. One or more taints are applied to a node; this marks that the node should not accept any pods that do not tolerate the taints. All these Kubernetes concepts are presented in a graphical form in Figure 12.

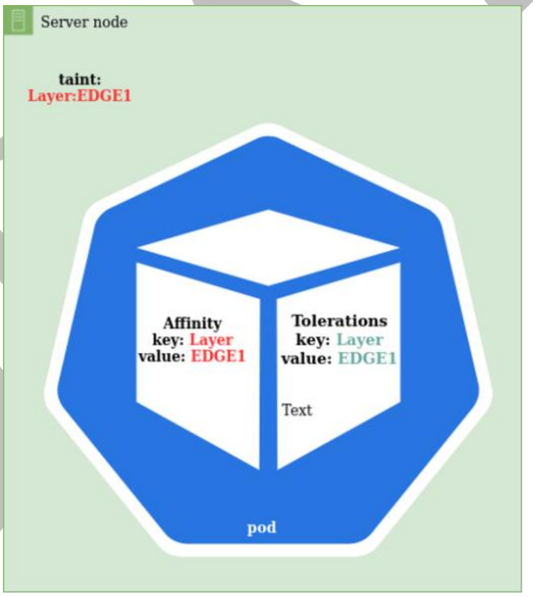


Figure 12: Kubernetes Affinity, Taints and Tolerations concepts in a node.

3.1.4.3 MARVEL examples

For the MARVEL Kubernetes cluster to better serve the MARVEL use cases, we label nodes of each layer (Fog, Edge) in order to assign pods. So as an example, for the GRN Edge host machine after the installation of the Kubernetes tools and the EdgeSec VPN, the host is part of the MARVEL Kubernetes cluster. Then node is labelled with the following command (Listing 32).

```
kubectl label nodes grnedge1 Layer=GRNEDGE1
```

Listing 32: Labelling example for the GRN Edge host machine

The affinity is applied to each pod by adding the following lines in the corresponding YAML file. (Listing 33).

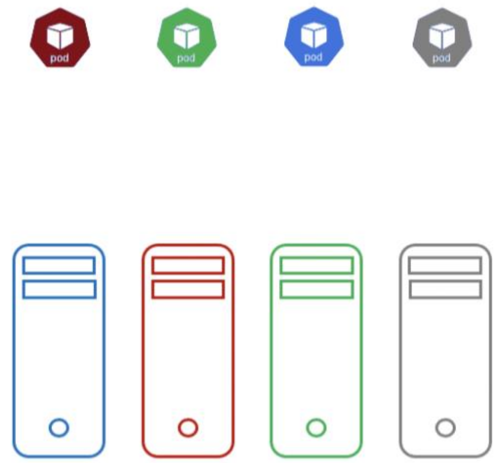
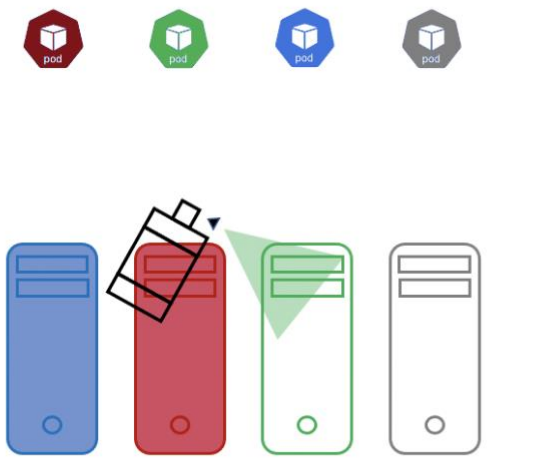
```
affinity:
  nodeAffinity:
   requiredDuringSchedulingIgnoredDuringExecution:
   nodeSelectorTerms:
   - matchExpressions:
     - key: Layer
       operator: In
       values:
       -  GRNEDGE1
```

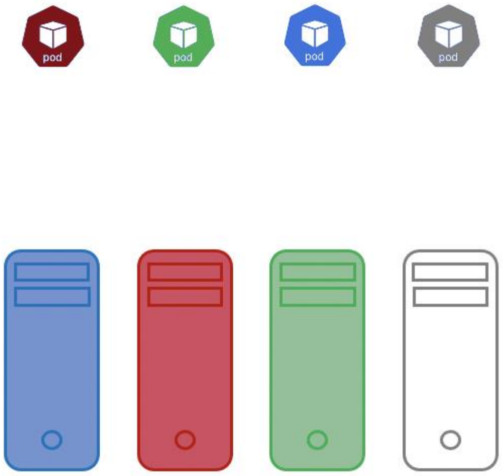
Listing 33: Affinity example for the GRN Edge host machine

So based on the aforementioned, the scheduler cannot schedule the Pod unless the key Layer is GRNEDGE1.

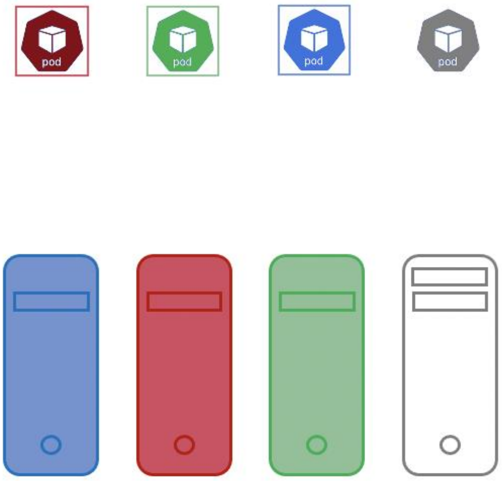
The full circle of taints and tolerations, and node affinity is described in the following steps (see Table 1):

Table 1: Applying taints/tolerations and node affinity in Kubernetes

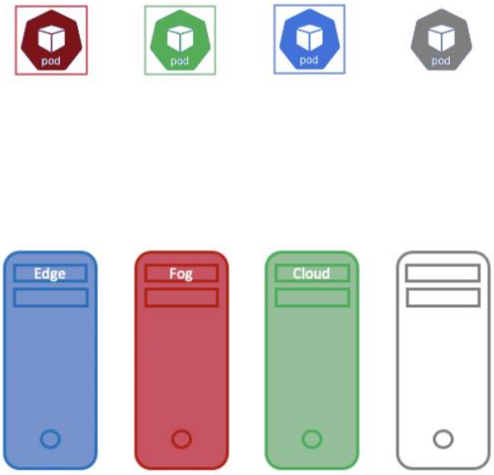
 **Step 1 -** This is a representation of pods and nodes of a Kubernetes Cluster.	 **Step 2 -** The different hosts are tainted in order to be used in different use cases.



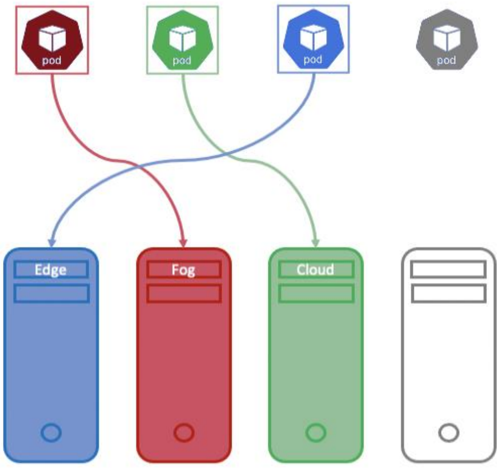
Step 3 - The nodes are tainted according to the different Layers.



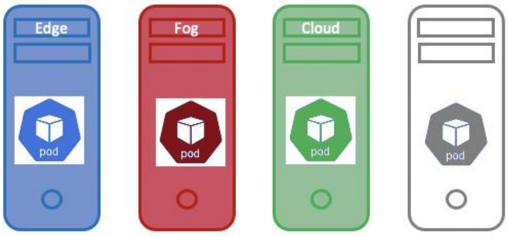
Step 4 - We add tolerations to the pods in order to be attracted to the tainted nodes.



Step 5 - The nodes are also labelled.



Step 6 - We use node affinity to limit the pods to the tainted and labelled nodes.



Step 7 - All pods are assigned to the desired nodes.

3.3 Future plans – Deployment optimisation

The current version of MARVdash allows for the deployment of AI and other MARVEL components at individual nodes of the Kubernetes cluster. The owner of each component is able to choose the cluster node for deployment. A combination of taints and tolerations, and affinity rules can be used together to completely dedicate nodes for specific pods. A set of pods corresponds to a given MARVEL component/application. Firstly, taints and tolerations are used to prevent pods with no selectors from being placed on tainted nodes, and then node affinity to prevent pods with tolerations from being placed on unlabelled nodes. However, the choice of the execution environment of a component is made manually, altering the values attributes of tolerations and affinity in the YAML file of each component.

Our vision regarding MARVdash is the selection of the deployment target to be made in an automated way based on the resource availability of nodes at each of the three layers of the MARVEL E2F2C framework. This new version of MARVdash will not demand declarative instructions about the target deployment node for each component. The component owners will just have to describe what are the resource requirements of their components, and MARVdash will make the decision of the desired execution environment based on the given resource requirements of the component and the actual availability of devices, network, and resource consumption across the entire E2F2C framework.

Kubernetes deals with the deployment of pods with specific resource requirements. When an application (in the form of one or more pods) is deployed in a Kubernetes cluster, the Kubernetes scheduler selects a node for the corresponding pods to run on. Such a node has a certain capacity for each resource type that is available for the pods to be deployed. The Kubernetes scheduler ensures that the sum of the resource requests of the scheduled containers of the pods is less than the capacity of the node. Even if the actual memory or CPU resource usage on a node at any given time is low, the scheduler will not place a pod on that node if the capacity check fails. This is a safety mechanism for potential increases in resource usage - peaks.

In that way, we will achieve deployment optimisation, deciding where the processing should be made, and aiming at optimising the E2F2C distributed DL architectures.

4 Model optimisation for efficient inference

4.1 Centralised compression methods

4.1.1 DynHP

In this section, we present the DynHP methodology developed for compressing DNN models at training time. This means that the DNN is incrementally trained and compressed at the same time. Moreover, the DynHP procedure can operate at a fixed memory budget. Such an approach enables, at least in principle, the possibility of operating the compression on resource-constrained devices.

The kind of compression that DynHP operates is pruning. These two terms will be used interchangeably in the following. Specifically, with the term compression, we refer to the identification of parameters or groups of parameters that can be zeroed without affecting significantly the inference performance of the DNN. Note that zeroing the parameters of a DNN does not reduce its memory footprint or the number of FLOPs. To achieve the actual reduction, it is necessary to redefine the network without the zeroed parameters. This operation is not part of the pruning methodology, and it is orthogonal to the specific methodology. Structured pruning is a type of pruning suitable for such kind of compression results.

Briefly, in structured pruning, the idea is to prune groups of parameters. This is the case for pruning some of the rows of the matrix representing a fully connected layer or some convolutional filters of a convolutional layer. The alternative approach to structured pruning is unstructured pruning where the parameters are pruned in a scattered fashion. The former is more suitable for both effective memory reduction at compute time and communication time since the model has a smaller number of parameters. The latter, instead, falls in the category of sparsification techniques and it is beneficial, for example, only for the efficient communication of models over the network (assuming the application of a lossless compression scheme that exploits the sparse representation of the weight matrices).

The other distinction needed to describe DynHP regards the kind of pruning: Soft Pruning (SP) vs Hard Pruning (HP). In SP, all the (groups of) parameters can be turned on and off during subsequent training epochs. This kind of pruning eases the training process because if a set of parameters was wrongly zeroed during the process, can be restored afterwards. On the other hand, HP is a one-way procedure. Precisely when a (group of) parameter(s) is turned off (i.e., zeroed), it can never be restored afterwards. With SP the actual removal of the unnecessary parameters can only happen at the end of the training, while with HP can be done incrementally during the process. DynHP belongs to the HP family.

4.1.1.1 Background on Soft pruning

In the following, we first introduce the notation, and then we discuss the SOTA techniques for soft pruning neural networks (during training) [6] that inspired DynHP. For the sake of clarity, we report only the details that are necessary to make this description self-contained.

We assume to have a dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N} \in \mathbb{R}^{N\times d+k}$ containing N i.i.d. d-dimensional observations $x_i \in \mathcal{X} \subseteq \mathbb{R}^d$, each one accompanied by a label $y_i \in \mathcal{Y} \subseteq \mathbb{R}^k$. Note that we target supervised learning problems with one or more labels per observation. The neural network model with weights $\boldsymbol{\omega}$ is denoted by the function $h: \mathbb{R}^d \to \mathbb{R}^k$. Additionally, in the following, we refer to the set of neurons with the symbol $\boldsymbol{\theta}$. Let $\ell: \mathbb{R}^k \times \mathbb{R}^k \to \mathbb{R}$ be the loss function used to evaluate the prediction accuracy of the model h. The operator $\odot$ is the Hadamard product, i.e., the element-wise product between vectors or matrices. Finally, let us denote with $\|\cdot\|_p$ the generic p-norm.

The loss function has the following form:

$$\boldsymbol{\mathcal{L}(\omega) = \frac{1}{N}\sum_{i=1}^{N} \ell\,(h(x_i;\omega), y_i) + \lambda\|\omega\|_0}$$

Equation **1**

where the first component of $\mathcal{L}(\boldsymbol{\omega})$ is the average loss of model h over the dataset, and the second component is the norm L_0 acting as a regulariser tuned with the $0 < \lambda < 1$ parameter.

Since the L_0-norm counts the number of non-zero parameters of the neural network, using it as a regularisation term drives the learning algorithm towards solutions having a small number of connections whose weight is non-zero. However, the L_0-norm of the weights is not a differentiable function, which prevents the usage of any gradient-based optimisation method from training the neural network.

To overcome the above problems, Louizos *et al.* [6] propose to approximate the L_0-norm with an equivalent and differentiable function. They propose a re-parametrisation of $\boldsymbol{\omega}$ such that each parameter ω_j (with $1 \leq \mathrm{j} \leq |\boldsymbol{\omega}|$), is defined as follows:

$$\omega_j = \widetilde{\omega_j} z_j$$

Equation **2**

where $z_j \in \{0,1\}$ is a binary gate that controls the activation of the j-th parameter. Therefore, the L_0-norm becomes:

$$\|\ \boldsymbol{\omega}\ \|_0 = \sum_{j=1}^{|\boldsymbol{\omega}|} z_j$$

Equation **3**

The main intuition is to model the gates as Bernoulli random variables:

$$q(z_j|\pi_j) = \mathrm{Bern}(\pi_j)$$

Precisely, each binary gate z_j has a probability π_j of being active. Adopting a probabilistic representation for the gates means that weights $\boldsymbol{\omega}$ and the loss value $\ell(,)$ become random variables. Therefore, the objective function expressed in Equation 1 needs to become an average:

$$\mathcal{L}(\widetilde{\boldsymbol{\omega}}, \boldsymbol{\pi}) = \mathbb{E}_{q(\mathbf{z}|\boldsymbol{\omega})}\left[\frac{1}{N}\sum_{i=1}^{N} \ell\,(h(x_i; \widetilde{\boldsymbol{\omega}} \odot \mathbf{z}), y_i)\right] + \lambda \sum_{j=1}^{|\boldsymbol{\omega}|} \pi_j$$

$$\widetilde{\boldsymbol{\omega}}^*, \boldsymbol{\pi}^* = \mathrm{argmin}_{\widetilde{\boldsymbol{\omega}}, \boldsymbol{\pi}} \mathcal{L}(\widetilde{\boldsymbol{\omega}}, \boldsymbol{\pi})$$

Equation **4**

In this way, the learning process is affected by the number of active binary gates and, in particular, we are minimising at the same time both the generalisation error and the sum of the probabilities of the gates. However, in this form, the function $\mathcal{L}(\tilde{\boldsymbol{\omega}},\boldsymbol{\pi})$ is not yet suitable for efficient gradient computation since the Bernoulli distribution is a discrete function and, consequently, it prevents the smoothness of $\mathcal{L}(\tilde{\boldsymbol{\omega}},\boldsymbol{\pi})$. The solution proposed is to substitute the Bernoulli distribution used for modelling the gates with the Hard Concrete Distribution which is continuous and differentiable approximation. Skipping all the technical steps, the final and differentiable version of $\mathcal{L}(\tilde{\boldsymbol{\omega}},\boldsymbol{\pi})$, denoted as $\mathcal{R}(\tilde{\boldsymbol{\omega}},\boldsymbol{\phi})$ has the following form:

$$\mathcal{R}(\tilde{\boldsymbol{\omega}},\boldsymbol{\phi}) = \frac{1}{T}\sum_{l=1}^{T}\left(\frac{1}{N}\left(\sum_{i=1}^{N}\ell\left(h\left(x_i;\tilde{\boldsymbol{\omega}}\odot \mathbf{z}^{(t)}\right),y_i\right)\right)+\lambda\sum_{j=1}^{|\boldsymbol{\omega}|}\left(1-Q_{s_j}\left(0|\phi_j\right)\right)\right)$$

Equation **5**

where $Q_{s_j}\left(0|\phi_j\right)$ is the Cumulative Distribution Function (CDF) of the Hard Concrete Distribution. Equation 5 assumes that training occurs over T epochs, and in each epoch, a random draw for the activation gates is used. The first part of the equation is thus the average over the samples of the average losses obtained at each round, which is a standard estimator for the average value of the average loss. The second part is the average value of the number of non-zero gates. We minimise the objective function with respect to both $\tilde{\boldsymbol{\omega}}$ and $\boldsymbol{\phi}$, meaning that we learn, at the same time, the parameters $\tilde{\boldsymbol{\omega}}$ and how many of them are, on average, useful for the good predictions ($\boldsymbol{\phi}$).

4.1.1.2 Incremental Hard Pruning

Our *Hard Pruning* mechanism is based on a "one-way-only strategy" aimed at identifying and removing the less useful neurons together with all their inward and outward connections. To this end, we exploit the weights gating mechanism discussed above and collect detailed statistics on gates activation during a fixed observation time window, e.g., a training epoch. At the end of each epoch, we compute the average activation rate $a_j \in [0,1]$ for each gate j, and we use such values to identify the least active neurons to be pruned.

More formally, let $\mathbf{Z} = \{0,1\}_{j=1}^{|\boldsymbol{\theta}|}$ be a vector that stores the binary information regarding the status (i.e., active/inactive) of each neuron of a layer l.[14] $\mathbf{Z}$ is updated at the end of each epoch to always record the active neurons in the layer. Moreover, let $\mathbf{a}$ be a vector recording the activation rate for each neuron in the layer during an epoch. $\mathbf{a}$ is computed as follows:

$$\mathbf{a} = \frac{1}{E}\sum_{e=1}^{E}\mathbf{z}_e$$

where $\mathbf{z}$ is the number of times the random gates were active during an epoch and E is the total number of gate's state observations (i.e., active or inactive) during a training epoch. To identify the less active neurons in the layer, we use a hard thresholding function $g(\cdot,\gamma)$ with fixed threshold γ:

[14] For the sake of clarity, the description refers to a single layer but its extension to all the layers of the neural network is straightforward.

$$g(x,\gamma) = \begin{cases} 1 & \text{if } x \geq \gamma \\ 0 & \text{otherwise} \end{cases}$$

By applying element-wise function $g(\cdot,\gamma)$ to vector **a,** we obtain a binary vector

$$\hat{\mathbf{z}} = g(\mathbf{a},\gamma)$$

that identifies the most active neurons in the layer. At the end of the epoch, we use this information to update the status of the neurons stored in **Z**:

$$\boldsymbol{Z} = \boldsymbol{Z} \odot \hat{\mathbf{z}}$$

In addition, vector **Z** is used to create a binary matrix M. M is finally used to set to zero all the weights corresponding to the deactivated neurons. This HP step is repeated at the end of each epoch for all the layers of the network.

4.1.1.3 Dynamic Batch Sizing

We now present our *Dynamic Hard Pruning* technique. As one might expect, our HP technique results in an effective reduction of the size of the neural network as training progresses, but it has a significant impact on the convergence of the training process. The worse performance is mostly due to the interplay between the hard pruning and the training processes. We contrast the performance degradation due to hard pruning by adapting the size of the minibatches as the training progresses. In this way, we obtain a double benefit, because with a single parameter to tune (i.e., the growth rate of the mini-batches, as we will explain in the following section), we control both the speed of convergence and the total amount of memory used by the learning process. We dynamically regulate the batch size according to the *relative variance* (or Variance-To-Mean Ratio) of the gradients. Formally, let S be the variance estimation of the gradients for the current mini-batch and F the value of the loss function for the current mini-batch. At the end of each gradient computation, the new size b of the mini-batch is computed as:

$$b \quad = \quad b + \left\lfloor (1-\alpha_{bs}) \frac{\parallel S \parallel_1}{F} \right\rfloor$$

where $\alpha_{bs}, \in [0,1]$ is a smoothing parameter used to drive the batch size according to the observed variance on the gradients. However, through this method, the batch size can indefinitely grow and, especially in resource-constrained devices, where memory is limited, this effect might prevent the successful training of the neural network.

To overcome this problem, we adopt the following procedure. At the end of each training epoch, after having pruned the network, we compute the memory available for the growth of the i-th mini-batch size as the difference between the memory budget ($\mathcal{C}$) and the current memory occupation of the network $\mathcal{N}_i$.

$$\mathcal{B}_i = \mathcal{C} - \mathcal{N}_i$$

We impose the maximum increase of the mini-batch to be

$$\varDelta\mathcal{B}_i = b_i + (\mathcal{C} - \mathcal{N}_i)$$

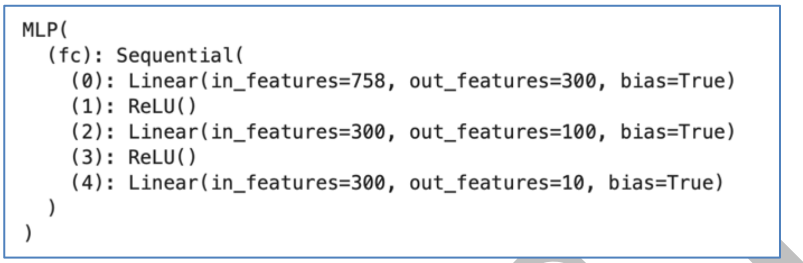
Denoting with $\tilde{b}_{i+1}$ the "candidate" size of the mini batch and taking into account the above maximum size limitation, we obtain that the mini-batch size at epoch $i+1$ is:

$$b_{i+1} = min\left(\varDelta\mathcal{B}_i, \tilde{b}_{i+1}\right).$$

Note that the network pruning is performed at the end of the epoch; therefore, at the beginning of the next epoch, additional free memory might be available to grow the size of mini-batches, i.e., $\mathcal{B}_{i+1} \geq \mathcal{B}_i$ always holds.

4.1.1.4 Model definition and available models

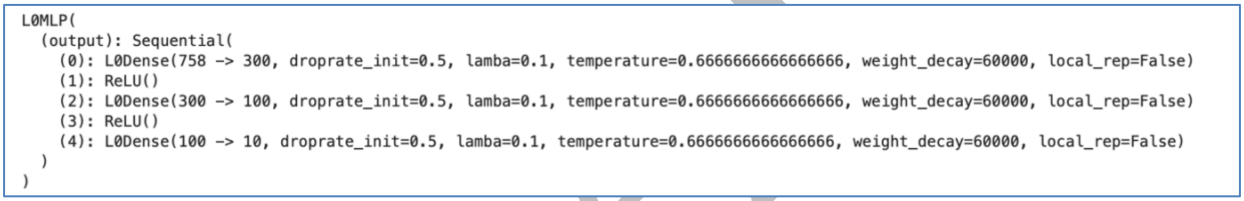
In order to apply the DynHP procedure to a DNN model, it is necessary to define the model using the DynHP primitives. For example, let us consider the Pytorch structure a simple Multilayer Perceptron as reported in the figure:



```
MLP(
  (fc): Sequential(
    (0): Linear(in_features=758, out_features=300, bias=True)
    (1): ReLU()
    (2): Linear(in_features=300, out_features=100, bias=True)
    (3): ReLU()
    (4): Linear(in_features=300, out_features=10, bias=True)
  )
)
```

Figure 13: Standard MLP definition

The corresponding definition using the DynHP primitives is as follows:



```
L0MLP(
  (output): Sequential(
    (0): L0Dense(758 -> 300, droprate_init=0.5, lamba=0.1, temperature=0.6666666666666666, weight_decay=60000, local_rep=False)
    (1): ReLU()
    (2): L0Dense(300 -> 100, droprate_init=0.5, lamba=0.1, temperature=0.6666666666666666, weight_decay=60000, local_rep=False)
    (3): ReLU()
    (4): L0Dense(100 -> 10, droprate_init=0.5, lamba=0.1, temperature=0.6666666666666666, weight_decay=60000, local_rep=False)
  )
)
```

Figure 14: MLP definition using DynHP primitives

Note that the difference between the modules used to build the MLP's layers. In the latter the Linear module is substituted by the L0Dense module which provides the functionality of a fully connected layer with, in addition, the parameters to train the gates discussed in the previous section. The same hold for the convolutional layers, i.e., the Pytorch Conv2d layer is substituted by a L0Conv2d layer which, similarly to the L0Dense, provides the additional learning parameter for the gates and all the machinery to train it.

Therefore, in order to define model compliant with the DynHP procedure, it is mandatory to use the corresponding L0* modules.

At the moment, the DynHP provides the following L0-topologies:

- MLP
- ResNet-28-1 and WideResNet-28-10
- VGG family

4.1.1.5 Performance evaluation

DynHP has been tested on three SOTA topologies (i.e., MLP, ResNet28-1) on three benchmark datasets (i.e., Modified National Institute of Standards and Technology (MNIST), Fashion-MNIST and CIFAR-10).

Table 2: Dataset description

Dataset	# Images		Image size (# pixels)	# channels
	Training Set	Test Set		
MNIST	60,000	10,000	28×28	1
Fashion-MNIST	60,000	10,000	28×28	1
CIFAR-10	50,000	10,000	32×32	3

We report in Table 3 the performance evaluation on MNIST. We evaluated the sensitivity of DynHP to the hyper-parameter α_{bs} (which regulates the growth factor of the batch size during training) w.r.t, the accuracy expressed in terms of misclassification error, the final model size obtained after the training and pruning process and the total memory occupation in the process. The baseline for both Hard Pruning and Dynamic HP is the Soft pruning method introduced in the Section 4.1.1.1.

Table 3: Performance on MNIST

Method	α_{bs}	Misclassification Error	Model Size	Tot. Memory Usage
		(%)	(MBytes)	(GBytes)
SP ($bs = 512$)	–	1.42 (–)	1.041 (–)	5.219 (–)
HP ($bs = 100$)	–	1.41 (−0.01%)	0.206 (−88%)	1.185 (−77%)
DynHP	0.971	1.50 (+0.08%)	0.572 (−45%)	5.165 (−1%)
DynHP	0.972	1.35 (−0.07%)	0.398 (−62%)	2.736 (−48%)
DynHP	0.973	1.40 (−0.02%)	0.205 (−80%)	1.279 (−75%)
DynHP	0.974	1.45 (+0.03%)	0.217 (−79%)	1.361 (−74%)
DynHP	0.975	1.43 (+0.01%)	0.178 (−83%)	0.980 (−81%)

With both HP and DynHP is possible to prune up to 88% of parameters with negligible performance degradation. On Fashion-MNIST, the results are in line with the previous ones.

Table 4: Performance on Fashion-MNIST

Method	α_{bs}	Misclassification Error	Model Size	Tot. Memory Usage
		(%)	(%)	(GBytes)
SP ($bs = 128$)	–	9.96 (–)	1.041 (–)	2.866 (–)
HP ($bs = 128$)	–	10.20 (+0.24%)	0.236 (−77%)	1.377 (−52%)
DynHP	0.979	9.97 (+0.01%)	0.430 (−59%)	2.874 (0%)
DynHP	0.980	9.64 (−0.32%)	0.415 (−60%)	2.874 (0%)
DynHP	0.981	9.96 (−0.00%)	0.418 (−60%)	2.874 (0%)
DynHP	0.982	9.77 (−0.19%)	0.399 (−62%)	2.874 (0%)
DynHP	0.983	10.21 (+0.25%)	0.407 (−61%)	2.872 (0%)
DynHP	0.984	10.12 (+0,16%)	0.405 (−61%)	2.872 (0%)
DynHP	0.985	10.06 (+0.10%)	0.410 (−61%)	2.872 (0%)
DynHP	0.986	9.93 (−0.03%)	0.386 (−63%)	2.858 (0%)
DynHP	0.987	10.08 (+0.12%)	0.252 (−76%)	2.573 (−10%)
DynHP	0.988	10.23 (+0.27%)	0.135 (−87%)	0.829 (−71%)

DynHP	0.989	10.50 (+0.54%)	0.124 (−88%)	0.611 (−79%)

Finally, using ResNet-28-1on CIFAR-10 the results are as follows:

Table 5: Performance on CIFAR-10

Method	α_{bs}	Misclassification Error	Model Size	Tot. Memory Usage
		(%)	(%)	(MBytes)
SP ($bs = 256$)	–	7.84 (–)	1.41 (–)	882.00 (–)
HP ($bs = 256$)	–	11.12 (+3.58%)	1.28 (−9%)	856.70 (−3%)
DynHP	0.71	15.46 (+7.62%)	1.02 (−28%)	756.53 (−14%)
DynHP	0.73	14.36 (+6.52%)	1.00 (−29%)	780.16 (−12%)
DynHP	0.75	11.21 (+3.37%)	1.01 (−28%)	799.75 (−9%)
DynHP	0.77	13.20 (+5.35%)	0.99 (−30%)	763.14 (−13%)
DynHP	0.79	11.13 (+3.29%)	0.98 (−30%)	726.93 (−18%)
DynHP	0.81	12.02 (+4.18%)	0.99 (−30%)	749.39 (−15%)
DynHP	0.83	11.42 (+3.58%)	0.97 (−31%)	693.06 (−21%)
DynHP	0.85	12.36 (+4.52%)	0.97 (−31%)	637.03 (−28%)
DynHP	0.87	12.16 (+4.32%)	0.95 (−33%)	544.30 (−38%)
DynHP	0.89	11.48 (+3.64%)	0.94 (−33%)	522.12 (−41%)

4.1.2 AVCC Compression through DynHP

In this section, we report the integration activity performed to apply DynHP compression to the AVCC model provided by AU.

First, the AVCC model was initially developed under the Tensorflow Framework. This required an initial porting effort to implement the AVCC model and training procedure using the Pytorch framework.

Specifically, the AVCC version considered here is the one that outputs a heatmap. We focused on this version because it represents the backbone for the one that outputs the actual crowd-counting (the number of heads in the input frame). From now on we will refer to the original uncompressed outputs a heatmap as AVCC while we will refer to the one defined using the DynHP primitives as L0AVCC.

We recall that AVCC backbone is composed of three blocks: the audio-block, the video-block, and the fusion-block.

- In the video-block, the video frames are processed by the first 13 layers of a VGG16 network.
- In the audio-block, the audio frames are processed by the VGGish network without the final classifier.
- The fusion-block processes at the same time the outputs of the audio-block and video-block and outputs a heatmap.

For the definition of the equivalent L0AVCC topology, we adopted the following procedure.

- We redefined the video-block where the VGG16 model has been substituted by the equivalent L0VGG16. As in the original version, we keep only the first 13 layers.

- We redefined the audio-block where the VGGish model has been substituted by the equivalent L0VGGish. As in the original version, we keep only feature extraction backbone.
- The fusion block is left untouched.

This represents a first attempt but several strategies can be adopted. The reason for leaving the fusion block as in the original AVCC is because it contains BatchNormalization layers. Since it is still unclear how to compress layers just before batch normalisation without affecting the overall structure and performance, this investigation is left for future work.

The training and compression approach remains unchanged with respect to the original DynHP methodology.

4.2 Efficient federated methods

4.2.1 Federated Compression

In this section, we report the initial activity performed to extend the DynHP methodology beyond the centralised settings. Precisely, the original DynHP approach is designed to work under the assumption that the dataset used for training and compressing the model is fully available. In this activity, we wanted to take a step further, releasing such an assumption and considering that the data needed for training the algorithm is available at different physical locations. Moreover, such data for privacy/ownership constraints cannot be moved from the location where it has been generated or collected in the first place. Such a scenario falls in the category of Federated Learning, where several devices (from now on called clients) collaborate with each other by means of a central and coordinating entity (from now on called parameter server) to train a globally shared AI model.

Considering these assumptions, we started investigating if it was possible executing the DynHP methodology within the Federated Learning settings. Specifically, the goal of this initial activity is to come up with a compressed model trained in federated settings.

As in the typical federated settings we have two entities: (i) the client that trains the model on the local dataset and (ii) the server that collects and aggregates the information shared by the clients at each communication round.

Client side

The client executes the DynHP procedure. With respect to the centralised settings, this phase has no differences. A model is trained and compressed on the local dataset for a certain number of epochs. At the end of the local training, the clients obtain: (i) an updated version of the model weights ($\boldsymbol{\omega}$) and an updated version of the $\boldsymbol{\phi}$ parameters that control the activation probability of the gates used to prune the model. After the completion of the local training, each client sends both the model weights $\boldsymbol{\omega}$ and the $\boldsymbol{\phi}$ to the Parameter Server for the aggregation step.

Server Side

The Parameter server collects, at each communication round, the information sent by the clients. Formally each client k sends the following sets of parameters: ω_k, ϕ_k. The aggregation algorithm adopted in this preliminary activity is FedAvg, i.e., the parameter server computes the weighted average of the parameters to update the global model. Therefore, denoting the global model ω_G and the global pruning parameters ϕ_G, they are computed as follows:

$$\omega_G = \frac{1}{K}\sum_{i=1}^{K} \omega_k$$

$$\phi_G = \frac{1}{K}\sum_{i=1}^{K} \phi_k$$

Once the aggregation is completed, the updated set of parameters is sent to the clients for a new round of local training and compression.

Since this is a preliminary activity, we considered a very simple scenario: 2 clients with each one holding half of the MNIST dataset. Local datasets are id between each other. The federated learning process is synchronous, i.e., at each communication round, all the clients communicate their updates to the parameter server.

The local model is a L0MLP with the following topology 768-300-100-10. The loss function is CrossEntropy.

In the following, we show the trend over the epochs for the two clients: the training error, the Validation error, and the Test error. As we can see, even in presence of a pruning process that shuts down the neurons, both clients are able to learn during the epochs and reach state-of-the-art performance for this kind of model. Interestingly, we see the effect of the pruning process in the validation error, i.e., the temporary accuracy degradation is because the pruning process on the two clients was not yet in sync.

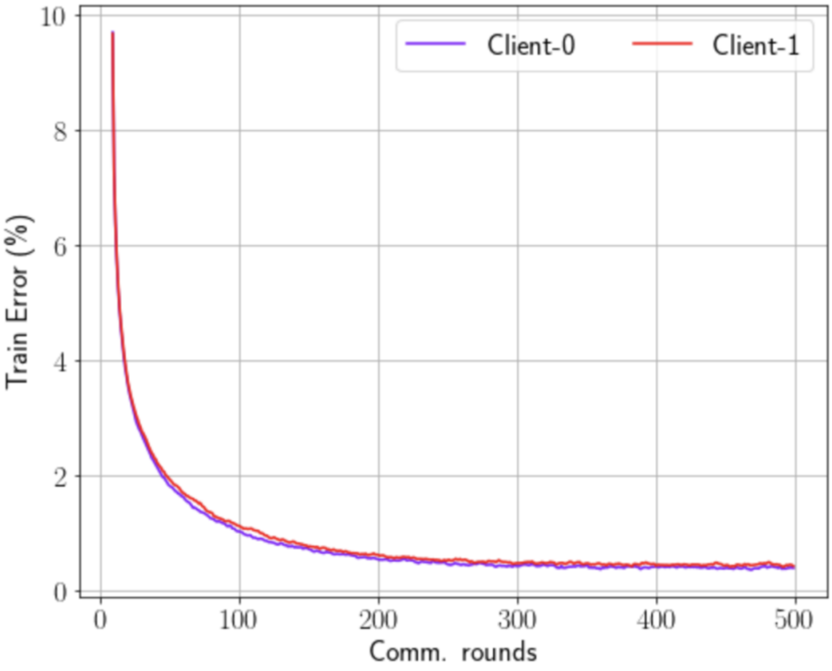


Figure 15: Training error over communication rounds

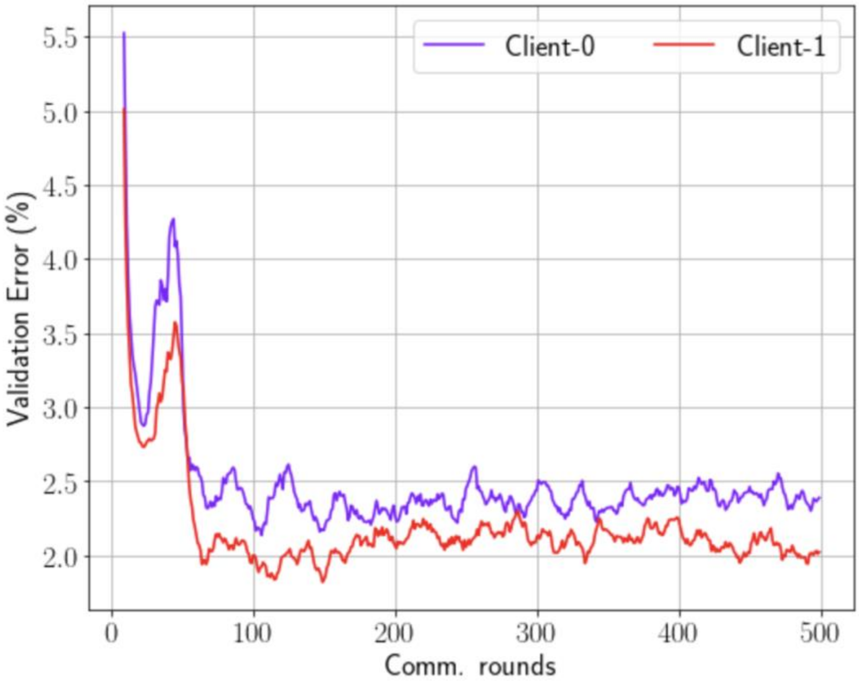


Figure 16: Validation Error over Comm. rounds

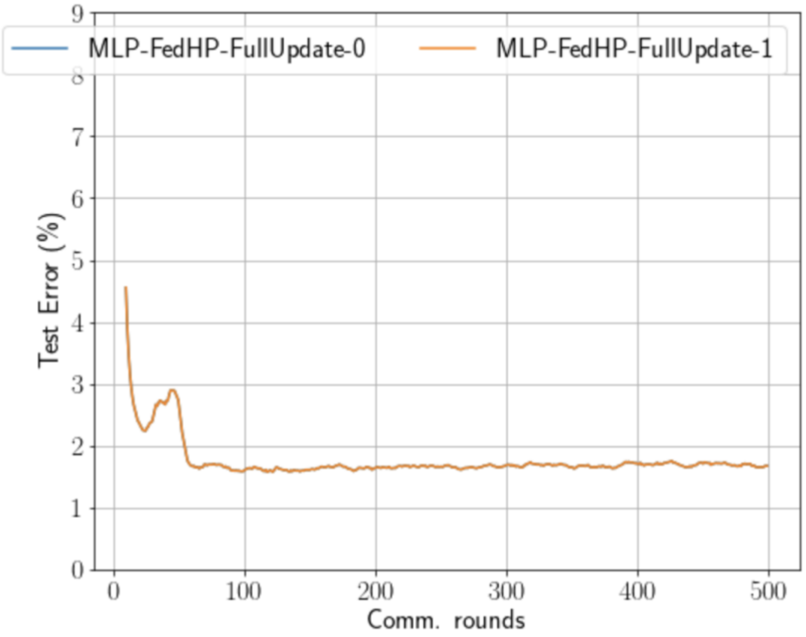


Figure 17: Test Error for clients 1 and 2 over comm. rounds

This aspect becomes evident looking at the evolution of the size of the network during the federated training process. In fact, for the first communication rounds the pruning process stales until it starts converging at both clients, as shown in Figure 18. The final model is 60% smaller than the initial one. These results are quite promising and make this kind of approach worth being investigated.

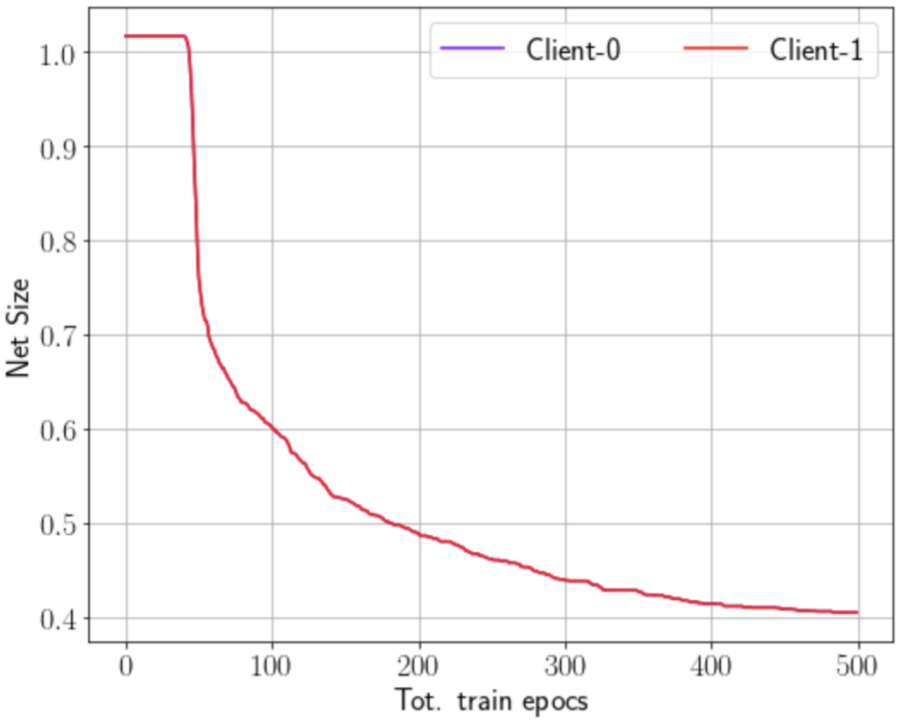


Figure 18: Model size during training and compression

5 KPIs

The section describes the relation of the main MARVEL components, associated with tasks T3.4 and T3.5, to the project KPIs and the corresponding component-related KPIs.

5.1 Project-related KPIs

- **KPI-O3-E2-1 CNR** (Model compression algorithms to achieve 70% compression rates, without a noticeable degradation of accuracy): The experiments reported in Section 4.1 using benchmark datasets, show that the target compression rate has been partially achieved for some of the topologies considered. Results report up 88% of compression for MLP and up to 30% for ResNET-28-1. An equivalent result is obtained also for VGG. It is worth noting that the compression is strongly dependent on the complexity of the learning task at hand. In the next months, the activity will focus on the improvement of the trade-off between compression and accuracy.

- **KPI-O3-E2-2** (Optimise performance (prediction accuracy, time-to-decision) of DL deployment by 20%): This KPI is linked with the distributed execution of DL tasks. Towards that end, the implemented E2F2C framework (Kubernetes cluster among with MARVdash dashboard on top of it) enables DL task distributed execution, taking into consideration the efficient use of execution resources. MARVdash contributes to the ability to match the task resource requirements to the various execution sites available in the MARVEL distributed environment. Consequently, it is possible to enable improvements both in performance, particularly time-to-decision, as well as in the sophistication of the DL models being deployed, thus enhancing prediction accuracy. The optimisation goal of this KPI will be achieved with the future functionality of MARVdash, planned to be implemented in the second half of the project lifetime.

- **KPI-O3-E2-3 (**Increase accuracy levels of real-time observations at the edge devices by 20%): This task is related to the deployment of the compressed models on edge devices, where the real-time requirements can be satisfied. Indirectly, it is possible to evaluate the FLOPs required to execute the model after compression and compare it with the FLOPs required by the original uncompressed model, by considering the compression of the model, as reported in **KPI-O3-E2-1.** Precisely, in the considered benchmarks, the FLOPs saved through compression is up to 36%.

- **iKPI-1.1** (At least three (3) tools for complex/federated/distributed systems handling extremely large volumes and streams of data): MARVdash contributes to this KPI indirectly, by enabling the instantiation of the first version of FedL component. FedL is scalable to a large number of FL clients and capable of handling data from multiple sites arriving in a streaming fashion. Newer version of FedL component may raise new requirements for MARVdash. The deployment of additional stream handling tools is planned for the next months.

- **iKPI-12.2 (**Increased performance in terms of response time, throughput, and reliability compared to a standard approach): Experiments with the FedL component, deployed through MARVdash show the potential improvements regarding the aforementioned metrics.

5.2 Component-related KPIs

- **MARVdash:** The component-related KPI for MARVdash focuses on usability. By that, we refer to the reduced effort to specify and automate component/service deployments. Another aspect of usability is user satisfaction when interacting with the MARVdash user interface. The baseline for this KPI is service deployment in a Kubernetes cluster without the functionality offered by the user-facing front end of MARVdash. A benchmarking process was followed and reported in D5.2[15] for the initial MARVdash assessment. According to the results of this process, participants rated MARVdash main functionalities with averages in the range of 4.75 (lowest) to 6.22 (highest). These values belong to the "Very good" category (one of them belongs to the "Excellent") of the corresponding qualitative assessment. Moreover, the user experience part of the assessment showed that MARVdash's means are above average compared with a large number of other products. Based on that result, MARVdash could be successful in the market.

- DynHP: The component-related KPI focuses on providing an interactive training. Since DynHP is a training methodology, it cannot be configured as a standalone service. The interaction with the user is required in order to configure the DNN training process, i.e., hyperparameter tuning, number of training epochs, etc. The DynHP component offers a Jupyter-lab environment through which it is possible to run and monitor the compression and training of a model on a specific dataset.

[15] "D5.2: Technical evaluation and progress against benchmarks – initial version," Project MARVEL, 2020. https://doi.org/10.5281/zenodo.6322699

6 Conclusion

In this document, we presented the process of creating the MARVEL E2F2C execution environment along with a dedicated dashboard (MARVdash) for implementing the interaction with the underlying environment, coordinating the execution of the data management platforms and other software components, and mediating external accesses to any service that needs to be exposed outside the MARVEL infrastructure. Moreover, a methodology for the compression of DNN model at training time, along with the actual application of this methodology to the AVCC model provided by AU was presented. This is the initial version of the document, while the final version of the benchmarking document will be prepared by the end of the project (M30), and it will contain functionalities that will be developed through the second half of the project's lifetime. Finally, the contribution to the MARVEL project and component-related KPIs was described.

7 References

[1] Kubernetes documentation <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>, accessed 5 June 2022.

[2] Open Container Initiative. Why are all these companies coming together? <https://www.opencontainers.org/faq#n7>, accessed 5 June 2022.

[3] W. Hummer, F. Rosenberg, F. Oliveira, and T. Eilam, Testing Idempotence for Infrastructure as Code. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 368–388.

[4] Y. Jiang and B. Adams, "Co-evolution of infrastructure and source code: An empirical study," in Proceedings of the 12th Working Conference on Mining Software Repositories, ser. MSR'15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 45–55.

[5] Cito, J., Schermann, G., Wittern, J. E., Leitner, P., Zumberi, S., & Gall, H. C. (2017, May). An empirical analysis of the docker container ecosystem on github. In 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR) (pp. 323-333). IEEE.

[6] Louizos, Welling, and Kingma, 'Learning Sparse Neural Networks through L_0 Regularization'. arXiv preprint arXiv:1712.01312.