TRANSPARENT APPLICATION DEPLOYMENT IN A SECURE, ACCELERATED AND COGNITIVE CLOUD CONTINUUM

Grant Agreement no. 101017168

Deliverable D5.2
Algorithmic Framework, Performance and Power Models

Programme:	H2020-ICT-2020-2
Project number:	101017168
Project acronym:	SERRANO
Start/End date:	01/01/2021 – 31/12/2023

Deliverable type:	Report
Related WP:	WP5
Responsible Editor:	ICCS
Due date:	31/03/2022
Actual submission date:	31/03/2022

Dissemination level:	Public
Revision:	FINAL

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 101017168

Revision History

Date	Editor	Status	Version	Changes
18.11.21	Panagiotis Kokkinos	Draft	0.1	Initial ToC
08.02.22	Ippokratis Sartzetakis	Draft	0.2	Initial content in Section 3
21.02.22	Kretsis Aristotelis	Draft	0.3	Initial content in Section 4
01.03.22	Gabriel Iuhasz	Draft	0.4	Add contribution in Section 6
07.03.22	Dmitry Khabi	Draft	0.5	Add contribution in Section 5
09.03.22	Konstantinos Kontodimas	Draft	0.6	Update content in Section 3.2
11.03.22	Panagiotis Kokkinos	Draft	0.7	Corrections, Add Sections 1,2,7, Ready for internal review
24.03.22	Panagiotis Kokkinos	Draft	0.8	Address internal review comments, include post review contributions
31.03.22	ICCS	Final	1.0	Final version for submission

Author List

Organization	Author
ICCS	Aristotelis Kretsis, Panagiotis Kokkinos, Polyzois Soumplis, Emmanouel Varvarigos
USTUTT/HLRS	Dmitry Khabi
INNOV	Stelios Pantelopoulos, Filia Filippou, Andreas Litke
UVT	Gabriel Iuhasz, Silviu Panica

Internal Reviewers

Ioannis Oroutzoglou, AUTH

Kamil Tokmakov, USTUTT/HLRS

Abstract: The deliverable D5.2 presents results of the activities that took place in the context of Task 5.2 “Multi-objective Optimization Mechanisms for Network Aware application Deployment and Service Assurance” and Task 5.4 “Energy and Resource Aware Flow Mapping” during the first iteration of the incremental implementation plan (M07-M15). These tasks aim to design and develop: (i) multi-objective network-aware resource allocation and service orchestration optimization algorithms, (ii) AI/ML-driven service assurance and re-optimization mechanisms, and (iii) energy and resource aware flow mappings.

Keywords: Resource Allocation, Multi-objective Optimization, AI/ML, Distributed Storage, Resource Optimization Toolkit, HPC Benchmark, HPC Services Service Assurance, Hyper-Parameter Optimization, Event Detection.

Disclaimer: *The information, documentation and figures available in this deliverable are written by the SERRANO Consortium partners under EC co-financing (project H2020-ICT-101017168) and do not necessarily reflect the view of the European Commission. The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The reader uses the information at his/her sole risk and liability.*

Table of Contents

List of Figures

List of Tables

Abbreviations

AI	Artificial intelligence
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
CCD	Compute Core Dies
CCX	CPU Complex
CU	Cost Unit
DNN	Deep Neural Network
DU	Data Unit
EDE	Event Detection Engine
FMA	Fused Multiply Add
GIPS	Giga Instructions Per Second
GPU	Graphics Processing Unit
HPC	High Performance Computing
HPO	Hyper-Parameter Optimization
ILP	Integer Linear Programming
IMC	Internal Memory Controller
IoT	Internet of Things
IoV	Internet of Vehicles
ITS	Intelligent Transportation System
JSON	JavaScript Object Notation
LIDAR	Light Detection and Ranging
MILP	Mixed-Integer Linear Programming
ML	Machine learning
MPI	Message Passing Interface
NBI	North Bound Interface
NN	Neural Network
NUMA	Non-Uniform Memory Access
OPC	on-premise component
PU	Period Unit
OSS	Object Storage Server
OST	Object Storage Target
QoS	Quality of Service
RAPL	Running Average Power Limit
REST	Representational State Transfer
ROT	Resource Optimization Toolkit
TU	Time Unit

1 Executive Summary

The deliverable is divided in seven main sections. Section 1 is this executive summary. Section 2 serves as an introduction to the document. It provides information about the document's purpose, structure, and audience to which it is mainly addressed.

Section 3 presents an initial set of resource allocation algorithms developed for the edge-cloud continuum. The first problem investigated is the efficient allocation of resources to serve data transfer-intensive, distributed Machine Learning (ML) training and inference operations. Next, storage allocation mechanisms are proposed that leverage the erasure coding technique to store data in a secure manner, while considering a number of different optimization criteria. The developed algorithms will be incorporated in the Resource Optimization Toolkit (ROT). Section 4 presents ROT architecture and interfaces.

Section 5 described the Excess cluster's setup that will be used to measure the energy efficiency of HPC services and to serve as a test platform for the SERRANO orchestrator. Also, a number of benchmark tests were executed to validate that the Excess cluster and its measurement system are well suited for evaluating the performance and energy efficiency of the Hawk supercomputer.

Section 6 presents the architecture of the Event Detection Engine (EDE) part of the Service Assurance component. A series of experiments presented for the creation of ML-based predictive models for the detection of anomalous behaviour.

Section 7 concludes the deliverable. It discusses the main findings and future work to be undertaken in the SERRANO project.

2 Introduction

2.1 Purpose of this document

The present deliverable (D5.2) presents the outcomes of Task 5.2 "Multi-objective Optimization Mechanisms for Network Aware Application Deployment and Service Assurance" and Task 5.5 "Energy and Resource Aware Flow Mapping", during the first iteration of the incremental implementation plan (M07-M15). T5.2 is associated with the development of multi-objective optimization algorithms for application deployment and data management in the distributed secure storage infrastructure of the SERRANO platform. It also includes the development of data-driven service assurance mechanisms and a framework to enable the execution of the developed optimization mechanisms. T5.4 is focused on the development of a framework that assists developers in incorporating the performance and power model functionality into the design and programming of their digital services, particularly within the SERRANO platform.

The objective of D5.2 is to build on the initial architecture of the SERRANO platform (Figure 1), as reported in the deliverable D2.3 in M9, towards the provision of the initial release of the SERRANO resource optimization algorithms, resource optimization toolkit, service assurance mechanisms and advanced power and performance models for deploying digital services within SERRANO platform.

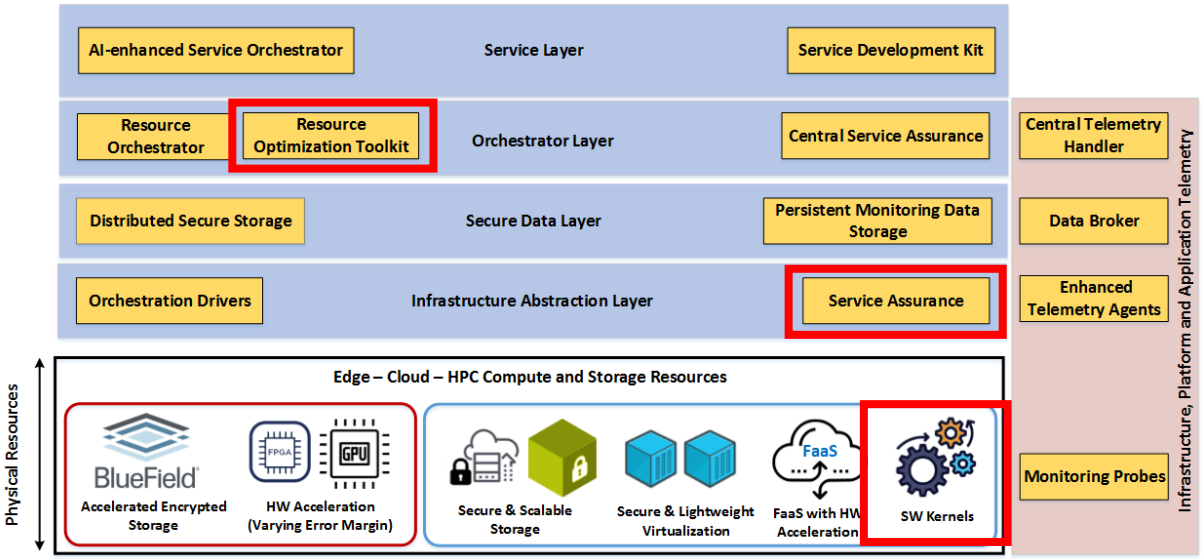

Figure 1: SERRANO high-level architecture

2.2 Document structure

The present deliverable is split into seven chapters:

- Executive Summary
- Introduction
- Algorithmic Framework
- Resource Optimization Toolkit
- Energy and Resource Aware Flow Mapping
- Service Assurance and Remediation
- Conclusions

2.3 Audience

This document is publicly available and should be of use to anyone interested in the initial description of the SERRANO mechanisms and algorithms related to multi-objective optimization and orchestration algorithms, data-driven service assurance and remediation mechanisms, HPC services and performance benchmarks.

3 Algorithmic Framework

The resource allocation problem in heterogeneous, dynamic, and multi-technology environments is rather complex due to the high number of conflicting objectives, necessitating the use of multi-objective optimization algorithms. Towards this direction, SERRANO within Task 5.2 develops a set of algorithms exploiting multi-objective optimization, AI/ML techniques, and heuristics to provide different trade-offs between optimality and complexity to satisfy the heterogeneous and strict applications' requirements efficiently. The most appropriate of the developed algorithms will be integrated into the Resource Optimization Toolkit (ROT) during the project's final stage. Next, we present the initial set of algorithms that developed during the first eight months of the task.

3.1 Resource Allocation for Distributed Machine Learning Training and Inference

Edge computing has emerged as a paradigm for local computing/processing tasks, reducing the distances (and latency) over which data transfers are made. Thus, an opportunity is presented for data transfer-intensive, distributed Machine Learning (ML) training and inference where the cloud can also play an assistive role. In this context, an important challenge is to allocate the required resources to carry out the processing. It is a complicated problem because of the various requirements (i.e., acceptable monetary cost, delay) of each ML job, as well as the features (i.e., processing, storage and bandwidth capabilities, location and cost) of the network resources. In this section we present a comprehensive solution for serving distributed ML jobs at the edge and/or at the cloud. It is comprehensive in the sense that it can be used to serve any kind of training and inference ML jobs. The conceptual description of this work with early results were submitted and accepted for publication at the ICC 2022 conference. We model the specific requirements of each ML job, and the features of the edge and cloud infrastructure resources. Next, we develop an Integer Linear Programming (ILP) algorithm to perform the resource allocation. We examine different scenarios (realistic processing and bandwidth monetary costs). We quantify tradeoffs related to performance and cost of edge/cloud bandwidth and processing resources. Our evaluations indicate that even though there are many parameters that determine the allocation, the processing costs at the different locations seem to have the most important role. The cloud bandwidth (b/w) costs can also be significant in certain scenarios. Finally, in certain examined cases, significant monetary benefits can be achieved through the collaboration of both edge and cloud resources when compared to using exclusively edge or cloud resources.

3.1.1 Introduction

In distributed ML training [1][2] or inference [3][4] the processing is performed on dedicated edge or cloud resources. This means that powerful computation resources are

employed. Moreover, the training or inference of an ML job is divided into a number of ML tasks that are executed in parallel in different equipment [3][5][6]. In distributed ML performed over edge and cloud resources, an important challenge is to allocate the most appropriate resources to serve each ML job with a certain objective (e.g., minimize the monetary cost). The challenge is similar to computation offloading [7]. However, it presents additional complications due to the requirements of distributed ML. First, a decision has to be made on whether a job will be served at the edge or at the cloud. The decision mainly depends on the acceptable cost and delay requirements of each job and the corresponding parameters of the edge and cloud resources. Then the suitable number and type of resources should be allocated according to the needs of the ML job (e.g., delay requirements, amount of data, type of ML algorithm). The problem requires modeling of the different contributors to the (bandwidth and processing) cost of a job that could be different at the edge and at the cloud. Moreover, there are different types of distributed ML architectures (e.g., different types of parallelism, communication architecture, and computation timing). All these factors and parameters have to be taken into account to formulate and solve the resource allocation problem.

In this section we investigate the resource allocation problem for distributed ML applications. More specifically, we consider a scenario where various devices are located at the edge of the network. The devices produce data that are used for ML training or inference at edge and/or cloud resources over an infinite time horizon. The goal is to assign the required resources (processing, memory, storage, bandwidth) for each machine learning job while optimizing certain metrics. We consider different scenarios related to edge/cloud costs, delay and performance. We introduce an Integer Linear Programming algorithm that solves the resource allocation problem. Finally, we compare the results, and extract interesting insights on various benefits and tradeoffs.

The main contributions of our work are:

1. We present a comprehensive solution that can serve both training and inference of distributed ML jobs. We appropriately model the resource allocation problem. We take into account different types (i.e., GPU models) of both edge and cloud resources, their computational performance, their bandwidth and processing monetary costs and their delay. The joint consideration of these optimization parameters provides a realistic modelling of the problem. Moreover, it allows for interesting insights on the various benefits and tradeoffs.
2. We present an Integer Linear Programming algorithm that solves the edge/cloud joint resource allocation problem. The objective is to minimize the monetary cost to serve all the ML jobs. The formulation is versatile, and can be used to allocate resources for various variants of distributed ML applications, training (such as model or data parallelism, and also all-reduce or aggregation servers) and inference. The algorithm can provide a timely solution to large instances of the problem.
3. We perform realistic simulation-based experiments to quantify the tradeoffs between edge and cloud resources for various bandwidth and processing costs of the edge vs cloud.

3.1.2 Problem Statement

We consider various ML scenarios consisting of certain devices (e.g., vehicles, IoT) at the edge (Figure 2). Each scenario could correspond to a different type of ML training or inference job, e.g., image recognition, anomaly or event detection, etc. There is an edge network close to the devices, and a more distant cloud. We assume that the ML *training* jobs are served either at the edge or at the cloud, depending on specific requirements that will be discussed in the next section. If the tasks of a training job were served both at the edge and at the cloud then the large variance of the communication time for the exchange of model information might significantly impact the performance of the algorithm. In any case, both training and inference jobs are completely offloaded to the network's resources, and are not processed at all on the devices. In future work we plan to extend the problem statement so as to include the possibility of (partial) execution of a job at the devices where the data are produced. In the following we describe some use cases in Internet of Things (IoT) and Internet of Vehicles (IoV).

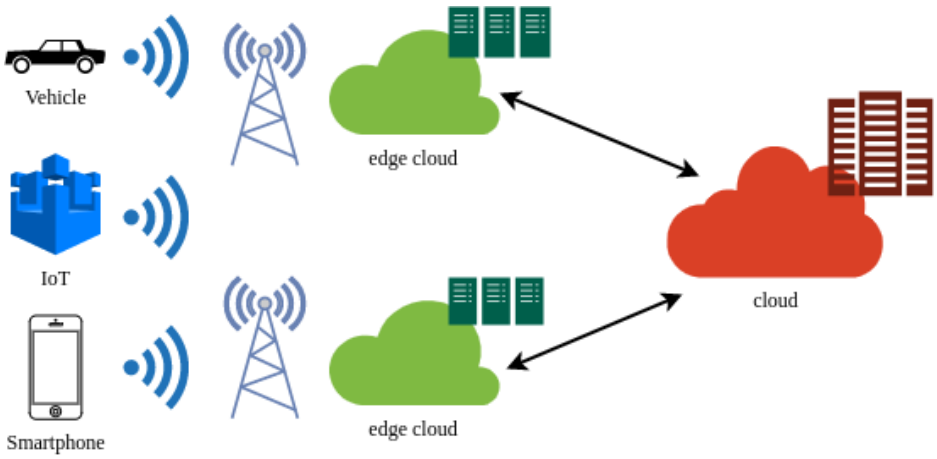

Figure 2: High-level architecture

Internet of Things

In recent years, IoT has spread across numerous applications and domains. Billions of devices are connected to the internet, gather data from their sensors and communicate with other devices. IoT applications range from smartphones, smart home devices and smart manufacturing in industries. IoT devices can have several different types of sensors: image (e.g., camera), voice (e.g., microphone), environmental (e.g., moisture sensor), mechanical (e.g., vibration sensor). Depending on the application and the type of the sensor, the data could be generated in high volume and streaming fashion. According to [8], the data produced by certain IoT sensors of a smart city could be approximately 8 GB/day.

The data of the IoT devices can be used in a large range of machine learning applications: image recognition (e.g., security related-face recognition, farming related - grapes disease detection) and other sectors such as smart electricity grids and healthcare. Also, in industrial IoT certain data from a manufacturing environment could predict and prevent mechanical failures. In these scenarios, the continuous learning is important in order for the model to be

adaptable to the environment and to other changes. Similarly, a smartphone may have a machine learning job related to voice recognition for digital assistants. Other examples of ML jobs can rely on data from accelerometers and gyro, e.g., to perform activity recognition.

Internet of Vehicles

Automotive industry is in the dawn of major transformation fueled by the developments in autonomous driving and the involvement of IT companies. At the same time, intelligent transportation systems (ITS) continuously evolve and apply novel communication protocols to provide safer and more efficient transportation. IoV recently emerged, leveraging the concept of IoT, as a network of "smart" vehicles that are interconnected and exchange data not only to enhance traffic safety and efficiency (e.g., mitigate traffic) but also to provide commercial infotainment (e.g., in the form of video and game streaming). IoV have significant processing and communication capabilities, supported by the network edge and cloud resources.

Moreover, future vehicles will have a large number of sensors. An autonomous vehicle can be equipped with many different sensors and cameras, such as Light Detection and Ranging (LIDAR), sonar and high-definition cameras. These sensors produce enormous amount of data. Each vehicle can generate 4 TB of data per day [9]. This amount of data will put significant stress on the network resources. In terms of communication resources, the transfer of such amount of data to the cloud is almost prohibited. Therefore, edge computing is expected to play an important role in IoV.

The data of the autonomous vehicles can be leveraged in many different distributed learning scenarios. In [10], a distributed learning vehicle routing algorithm is proposed. The algorithm adjusts vehicle routing in real time. This results in reduction of traffic congestion with the dynamic changes of the road environment. Another example is object detection and classification based on acquired charge-coupled device (CCD) and LIDAR data [11]. There have been several instances where an autonomous vehicle has misinterpreted an object in the environment. Therefore, the continuous training of object detection is important, given the variety and dynamicity of the environments that autonomous vehicles may navigate in. Similarly, an autonomous vehicle may require an urgent image recognition inference job. Finally, speech recognition of a driving assistance system is another scenario for distributed ML in IoV.

3.1.3 Problem Formulation

This section will formally define the problem to tackle the aforementioned distributed ML scenarios. The formulation is generic since it can be used for the above scenarios and different distributed ML architectures. We consider a number of devices that continuously produce data. These data are used for data parallel ML training or inference that will take place at the network's edge or cloud resources. Table 1 contains all the following notations. Each device u continuously produces data at an average rate of λ_u samples/sec. A set of a certain number and type of devices (or even a single device) and their data form an ML job j. A job can be further characterized as j_x for training and j_i for inference. The set J contains all

the ML jobs to be supported by a given infrastructure. The processing of an ML job j is divided into a set of distributed ML tasks T_j (or even one task, in case of e.g., lightweight inference) that are executed in respective workers (Figure 3). Each ML task t_{jk} is responsible for a subset D_{jk} of the entire dataset D_j of each job j. A device u_{jk} is related to the k^{th} ML task and belongs to the set of devices U_j of the j^{th} job. Regarding the training of each ML task t_{jk}, the loss function on its respective dataset is:

$$F_{jk}(w) = \frac{1}{|D_{jk}|} \sum_{l \in D_{jk}} f_l(w), \qquad (1)$$

where f is a per-sample loss function, $|\cdot|$ denotes the size of the set, w is the model parameter vector and l a training data sample. The training of a job j takes into account all of its tasks, and aims to find:

$$\min_w F_j(w) = \frac{\sum_k D_{jk} F_{jk}(w)}{D_j}. \qquad (2)$$

The time axis is divided in time periods of duration P_o. The symbol o represents ML applications with different time scale constraints. For example, we could have three different classes, one for training which typically requires more time and is usually not time critical, one for time-critical inference, and one for generic inference. The periods are asynchronously defined for each resource. The devices feeding an ML job either continuously (streaming) or in batches (at the end of a period) upload their data to the network's resources that run an ML algorithm. During a period, a resource performs training by processing a batch of data, consisting of data received at the previous period (i.e., until the current period begins). During each period, the worker node where task t_{jk} is assigned runs the ML algorithm to minimize its loss function, given by Eq. (1), and sends the new values of its parameters w to an aggregation node. The aggregation node updates the parameters by minimizing the overall loss function, given by equation (2), and returns the new values of the parameter vector w to the worker nodes, which incorporate them in their computations. We denote this roundtrip communication time by P_o^{comp} .

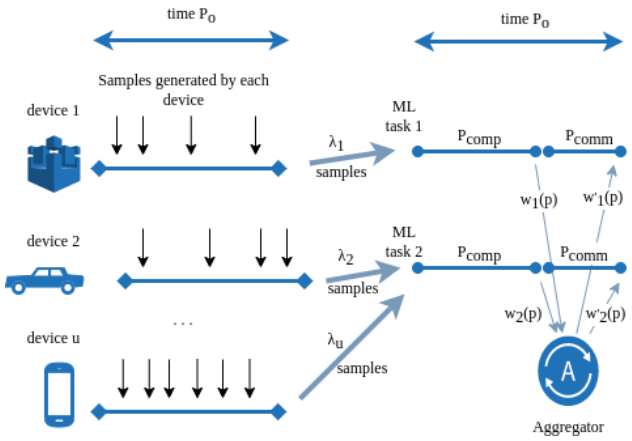

Figure 3: An example of an ML job with the timings of the training. Device u sends either continuously at rate λ_u or in batches of average size $P_o\lambda_u$ data for processing at a worker node assigned to it

Each resource continues to receive data from the devices that will be processed at the following period. At each time period P_o, device *u* produces and sends for training $s_u = P_o\lambda_u$ samples. Each ML task t_{jk} has a total $S_{jk} = \sum_{u \in U_{jk}} P_o\lambda_{\mathrm{u}}$ samples that have to be processed within period P_o. We require that the working node is assigned enough computation power to be able to process all the samples of an ML *training* job within a period of the duration P_o, otherwise the work generated within a period would be more than the work that can be completed within the same period and the system would be unstable. The same applies to the communication resources that should be allocated for transferring of the data from a device to the worker node assigned to it, which should also be sufficient. The communication resources needed between the worker nodes and the aggregator are small, as this transfer involves transmitting parameters (of small size) and not data samples. We also let $P_o^{comp} < P_o$ be the total computation time required to finish the training of a batch of data. Depending on the architecture of the distributed ML training algorithm, there could be different relationships between the timings P_o^{comm} and P_o^{comp}. Assuming that the workers first complete one computation round and then they send the model weights for averaging (i.e., no pipelining, so there is no overlapping of communication and computation times), then we have $P_o = P_o^{comp} + P_o^{comm}$. Thus, $P_o^{comm} = \omega P_o^{comp}$ for some ω<1. The actual communication overhead ω depends on various parameters such as the hardware configuration and the specific ML model [12]. If we assume pipelining, then the communication overhead can be reduced up to 95%, and a perfect overlap of computation and communication can be achieved [12]. The pipelining capability means that the ML algorithm is robust to asynchronism so that individual working nodes can continue processing samples and updating their weights, and can incorporate in their calculations the new values of the weights when they arrive from the aggregator (e.g., this would be easy to do if each worker node is responsible for updating a different set of weight parameters). In these cases, $P_o \approx P_o^{comp}$. In the case of inference, the communication overhead is either zero (in case of inference executed in just one resource) or generally much lower than that for the case of training (there is no communication of weights over many epochs like in training). Moreover, the efficient layer partitioning (model parallelism) of a Deep Neural Network (DNN) can result in even more reduced communication overhead [3].

3.1.3.1 Vector of Resource Requirements

In order to perform the updates, each ML task needs certain computation, storage and network resources. Each ML task t_{jk} running on a worker node has specific processing (GPU based), memory, storage, ingress bandwidth and possibly aggregator requirements that are directly related to the number of samples it receives and processes. These requirements are respectively described by a *vector of resource requirements* $R^{jk}(Z) = \left[G^{jkq}, M^{jk}, V^{jk}, B^{jk}, A^{jk}\right]$, where *G*, *M*, *V*, *B*, and *A* are parameters that determine the amount of processing, memory, storage, number of bits communicated (sample size) to the node and aggregators that each task requires for the specific ML application. As we allow processing units of different types, we use superscript *q* on *G* to refer to the specific model of processing unit taken from the set *Q* of all the available GPU models. The rationale for

introducing the resource requirements vector R^{jk} is that each ML task results in some specific processors to handle the samples, some specific memory, and specific bits to store and communicate the samples. Thus, it is natural to assume that the size of the resources of the different types (apart from the aggregator) is proportionally dependent on the number (or rate) of the samples with the proportionality constants that convert samples to requirements given by R^{jk}. Of course, the entries in R^{jk} depend on the type of ML application Z (e.g., parameters *G, M, V, B* will be different for image or voice recognition, etc. They also depend on the compression techniques used to encode an image or voice sample). More specifically, an ML task has processing workload $G^{jkq}(Z)S_{jk}$ (measured, e.g., in number of arithmetic operations or specific GPU hours) which has to be completed within a time period P_o. Assuming perfect pipelining of computation and communication, the task requires processing rate $G^{jkq}(Z)S_{jk}/P_o = G^{jkq}(Z)\sum_{u\in U_{jk}}\lambda_u$. If the overlapping is not perfect, then the processing rate should be increased by 1+ω, to account for the communication overhead (the same work has to be completed within a part of the period). An ML task similarly requires memory $M^{jk}(Z)S_{jk}/P_o = M^{jk}(Z)\sum_{u\in U_{jk}}\lambda_u/P_o$ and storage $V^{jk}(Z)\sum_{u\in U_{jk}}\lambda_u/P_o$. These values can be translated to specific resource units. More specifically, we assume that each resource can be assigned in units of predetermined granularity. E.g., one unit of memory could correspond to 1GB of RAM and a task could require say 10 GIPS (giga instructions per second) or 4 model *q* GPU units to process its workload. An aggregator unit could correspond to one CPU core. Moreover, the required ingress bandwidth B^{jk} can be derived as follows. Let β be the number of bits needed to represent a sample (β depends on the type of sample, e.g., picture, sentence, etc). The total ingress bandwidth required for the worker node running task t_{jk} is $B^{jk} = \beta\sum_{u\in U_{jk}}\lambda_u$. When assigning tasks to resources, we will use this total ingress bandwidth requirement as a criterion (ignoring the way it is subdivided among the communication channels leading to the specific devices).

Table 1: Important Notations

Symbol	Description
J	Set of all ML jobs
j_x, j_i	An ML training or inference job
T_j	Set of all ML tasks of job j
t_{jk}	An ML task of job *j*
u_{jk}	A device related to the k^{th} ML task of the j^{th} job
λ_u	The data (samples) production rate of a device *u*
N	The set of nodes of the edge network
n	A node of the edge network
P_o	The time period in which all training tasks are completed
P_{comm}	Communication time required to transfer the ML model weights
P_{comp}	Computation time required to complete an ML task
S_{jk}	Total samples that a task has to process within period $\boldsymbol{P_o}$
Z	Type of ML job (e.g., image recognition)
Q	The set of all the available GPU models
q	A specific GPU model

H_j	Required number of training epochs for each job *j*
C_E^q, C_C^q	The cost for an ML job to use model *q* processing edge/cloud res.
C_E^w, C_C^w	The b/w cost of edge or cloud
δ_E, δ_C	The propagation delay of a job if served to the edge or cloud
Δ_j	The maximum acceptable propagation delay of a job
$R^{tj} = [G^{tjq}, M^t, V^{tj}, B^{tj}, A^{tj}]$	Vector of required GPU, Memory, Storage, B/w res. of task $\boldsymbol{t^{jk}}$
$R_n^{Gq}, R_n^M, R_n^V, R_n^B, R_n^A$	GPU, memory, storage, incoming b/w, aggregator res. of node *n*
ξ_n^{jkq}, ξ_C^{jkq}	Binary ILP variable equal to 1 if task $\boldsymbol{t_{jk}}$ uses resource units and GPU model *q* at edge node *n,* or it is served at the cloud
e_{jx}, c_{jx}	Binary ILP variable equal to 1 if a training or inference job is served at the edge or at the cloud

3.1.3.2 Edge-Cloud Resource Infrastructure Model

Regarding the infrastructure on which the tasks are going to run, we consider an edge and a cloud network. The edge network consists of a set of nodes *N*. Each node has finite resources that can be used by the machine learning tasks. More specifically, each edge node *n* has finite number of R_n^{Gq} of GPU or CPU units of type *q*, R_n^M memory units, R_n^V storage units, R_n^B incoming bandwidth (b/w) units to receive the data from the devices and R_n^A aggregator units. Obviously, the sum of the required resources of all the tasks that are assigned to a node should not exceed the node's resources. The cloud network, on the other hand, is assumed to have infinite resources. A major difference between the edge and the cloud are the respective monetary processing and bandwidth costs. The cost to use a model *q* processing unit is defined as C_E^q in the edge and C_C^q at the cloud. The cost of ingress b/w is defined as C_E^{bw} in the edge and C_C^{bw} at the cloud. Finally, another difference between the edge and the cloud is propagation delay. Since the edge network is much closer to the devices, the propagation delay to the edge, denoted by δ_E, is expected to be significantly lower than the respective delay to the cloud, denoted by δ_C. Certain ML inference jobs may have stringent constraints on the maximum acceptable propagation delay (Δ_j). This should be taken into account by the resource allocation algorithm.

The goal of the resource allocation algorithm is to reserve the appropriate number of resources for the ML tasks while minimizing certain objectives and satisfying all the possible constraints.

3.1.3.3 Resource Allocation Algorithm

In this subsection we present the ILP algorithm responsible for the allocation of the resources. The algorithm gets certain inputs, and using some related constraints aims to allocate the network's resources (the variables of the algorithm), while satisfying the objective. The formulation assumes that there is one aggregation (parameter) server for each ML training job. It can be modified in a straightforward way to account for multiple aggregation servers or for all-reduce architectures. The algorithm provides a solution for a

given period P_o. Whenever the parameters change (e.g., a mobile dynamic scenario), the algorithm is re-executed to provide a new solution.

Inputs:

$N, R_n^{Gq}, R_n^M, R_n^V, R_n^B, R_n^A, J, T_j, R^{jk}, Q, C_E^q, C_E^{bw}, C_C^q, C_C^{bw}, \delta_C, \Delta_j$

Variables:

$\xi_n^{jkq}, \xi_C^{jkq}, e_{jx}, c_{jx}$

Objective:

The objective is to minimize the total cost to serve the jobs. The cost of each job depends on the amount of bandwidth (b/w), the model of GPU and whether it is served at the edge or the cloud:

$$min\left(\sum_j\left(\begin{array}{l}\sum_n\sum_{T_j}\xi_n^{jkq}\left(C_E^{bw}B^{jk}+C_E^G G^{jk}\right)\\+\sum_{T_j}\xi_c^{jkq}\left(C_C^{bw}B^{jk}+C_C^G G^{jk}\right)\end{array}\right)\right) \qquad (3)$$

Subject to:

- Each training job (and therefore all of its tasks) is served in either edge or cloud:

$$\forall j_x: e_{jx}+c_{jx}=1 \qquad (4)$$

- If the training job is served at the edge (if $e_{jx}=1$), then each one of its tasks should be served (in one node), and should use only one model of GPU, since e_{jx} will be equal to 1, and the tasks variables and GPU models are summed:

$$\forall j_x, \forall t_{jk}: \sum_{n\in N}\sum_q \xi_n^{jkq}=e_{jx} \qquad (5)$$

- If the training job is served at the cloud, then each task should use only one model of GPU:

$$\forall j_x, \forall t_{jk}: \sum_q \xi_c^{jkq}=c_{jx} \qquad (6)$$

- All the tasks of an inference job should be served once, at the edge and/or at the cloud and each task should use only one model of GPU:

$$\forall j_i, \forall t_{jk}: \sum_{n\in N}\sum_q \xi_n^{jkq}+\sum_q \xi_c^{jkq}=1 \qquad (7)$$

- Each edge node should have enough (#GPUs, memory, storage, bandwidth, aggregator) capacity to serve the assigned tasks. So, for all nodes we sum all the resources that a job could potentially use (ξ_n^{jkq}), and this sum should be less than the capacity of each node:

$$\forall n \in N, \forall q \in Q: \sum_j \sum_{t_{jk}} \xi_n^{jkq} G^{jkq} \leq R_n^{Gq}$$

$$\forall n \in N: \sum_q \sum_j \sum_{t_{jk}} \xi_n^{jkq} M^{jk} \leq R_n^M$$

$$\forall n \in N: \sum_q \sum_j \sum_{t_{jk}} \xi_n^{jkq} V^{jk} \leq R_n^V \qquad (8)$$

$$\forall n \in N: \sum_q \sum_j \sum_{t_{jk}} \xi_n^{jkq} B^{jk} \leq R_n^B$$

$$\forall n \in N: \sum_q \sum_{jx} \xi_n^{jkq} \leq R_n^A$$

- In order for an inference job to be served to the cloud, its maximum acceptable delay should be respected:

$$\forall j_i, \forall t_{jk} : \sum_q \xi_c^{jkq} \Delta_j \leq \delta_C \qquad (9)$$

In Eq. 3 the objective is to minimize the total cost of serving all the ML jobs. The first part of the equation refers to the cost of a job if it is served at an edge node *n*. The cloud cost is similar to the calculation of the edge cost, only without the *n* nodes. The cost consists of the b/w units B^{tj} of each task times the cost of b/w at the edge C_E^w, plus the model *q* processing units G^{tjq} of each task, times the cost of each processing unit C_E^q. The second part of the equation refers respectively to the cost of a job if it is served at the cloud. Eq. 4 ensures that all training jobs will be served, and they will be served either at the edge or at the cloud. Eq. 5 ensures that if a training job is served at the edge (e_j), then each one of its tasks will be served, will be served only once in one node and use only one model of GPU, as indicated by the double summation for all the edge nodes *n* and model *q* of GPUs. Eq. 6 similarly guarantees that if a training job is served at the cloud (c_j), then its tasks will be served once and use only one model of GPU. Eq. 4 together with Eq. 5 and Eq. 6 ensure that all the tasks of a training job are served in either the edge or cloud. Eq. 7 ensures that all the tasks of all the inference jobs will be served once at the edge and/or at the cloud. Eq. 8 constrain the sum of resources (processing, memory, storage, b/w and aggregators in case of training jobs) that all the tasks served at each edge node use, to be less or equal than the capacity of each edge node. Finally, Eq. 9 guarantees that should an inference job be served at the cloud (c_j), then the maximum acceptable delay of the job will be less than the delay of the cloud.

The ILP algorithm can serve the ML jobs at the edge or cloud with the objective to minimize the total cost while satisfying the constraints and the requirements of distributed ML. Since there is a large number of optimization parameters, the solution to the problem is not trivial. In the following section we examine various scenarios and evaluate the tradeoffs in each case.

3.1.4 Results

To evaluate our proposed resource allocation framework and quantify the edge-cloud cost relationships, we performed a number of simulation experiments. For all the parameters we assigned values that we consider realistic. We used a desktop computer with a quad-core CPU at 4 GHz with 16 GB RAM. We used Python and the Pyomo [13] optimization software to code the ILP and IBM CPLEX [14] to solve the problem.

3.1.4.1 Simulation parameters

We assumed a 20-node edge network with finite resources. The network could correspond to the edge facilities of a megacity. For example, in the New York metropolitan area, Google currently operates 8 edge nodes [15]. There are also several other service providers in the area, with their own equipment. In the immediate future further expansion of these facilities is almost certain. Thus, an edge network of 20 nodes seems realistic. We also assumed an abstract cloud with infinite resources. Each edge node has 5 racks. One rack is comprised of 10 servers, and 1 server has 4 GPUs. Thus, each edge node has 200 GPUs. For each edge node we also consider the following resources available exclusively for distributed ML purposes: 25 GB RAM, 10 TB of storage, 10 Tbps incoming bandwidth and 6000 CPU physical cores (that could correspond to approximately 100 CPUs).

We considered two simulation scenarios. One consists of a total of 100 training image recognition ML jobs. The other one consists of 100 inference image recognition ML jobs. This number of jobs could correspond to jobs from fleets of IoV coupled with jobs from networks of IoTs. We chose to run two different experiments so that the tradeoffs related to cost and other parameters will be clear for training and inference. The size Δ of each sample (image) of a job is chosen uniformly from the following set of integers: [0.4, 0.8, 1.2, 1.6, 2, 2.4] MBs / sample. The available GPU model was NVIDIA DGX-1 with 1 GPU V100 16G. The respective cost of this GPU at the cloud is $2.08/hour. The b/w cost to transfer data to the cloud is $0.01/GB. The respective pricings are taken from [16]. More specifically, the respective processing instance in Amazon is p3.2xlarge and the pricing corresponds to reserved instances. The required b/w of each task is derived by multiplying the generation rate of samples/sec by the size in MBs/sample and by the number of seconds of period P_o. This figure equals to the amount of data that have to be transferred within one period. The calculation of the required storage and memory is relatively trivial and does not play a significant (monetary) role in the resource allocation problem.

We examined a set of different parameters to evaluate the tradeoffs between processing-b/w cost at the edge and at the cloud. More specifically, we assumed different: i) edge vs

cloud bandwidth costs, ii) edge vs cloud processing costs, iii) number of epochs (in the case of training). According to [17] the edge's b/w costs can be approximately 0.1 times the cloud's. We therefore assumed that the edge b/w cost could be [0.5, 0,25, 0.1] times the cost to transfer the data to the cloud. Moreover, according to [18] the edge processing costs can be approximately 1.5 times the cloud processing costs. We therefore assumed that the processing costs at the edge could be [1.2, 1.5, 2] times more than that of the cloud. In the two following subsections we will describe some specific parameters related exclusively to training or inference.

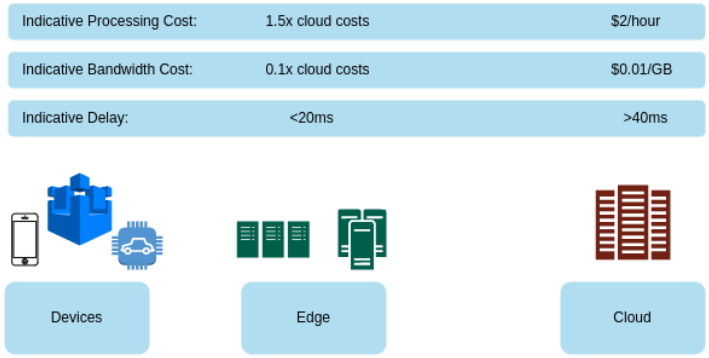

Figure 4: The processing and bw costs and the delay of the resources

Training Parameters

Each training job consists of either 2, 3, 4 or 5 ML tasks, uniformly distributed over all jobs. The sum of the data production rate of the devices of each task ($\sum_{e\in E_{jk}} \lambda_{e_{jk}}$) is 10 samples/sec. We consider that the duration of the training period is $P_o = 60$ seconds. The exact number of the ML tasks per job, the sum of the $\lambda_{e_{jk}}$ and P does not play an important role to the simulation and the resulting tradeoffs. They almost only affect the magnitude of the problem. The training performance Π^q of the GPU is Π^q =566 samples/sec. The assumed training performance is based on [19]. We assume that the training is fully pipelined, i.e., the computation and communication times fully overlap. To find the required number of GPU resources G^{jkq} per task, we first multiply the duration of the training period (P_o) by the number of samples/sec of the task (S_{jk}) and by the number of epochs H_{jk}. Then we divide by the performance in samples/sec Π^q within period P_o. The (rounded) result is the number of required GPUs for the ML task (Eq. 10).

$$G^{jkq} = \left\lceil \frac{P_o S_{jk} H_{jk}}{P_o \Pi^q} \right\rceil \qquad (10)$$

The required number of epochs can depend on factors, such as the type of the ML algorithm, the layers of the Neural Network, the desired accuracy, whether the training is continuous or not, and many other parameters. According to [20] the number of epochs required for certain ML benchmarks to reach the required accuracy can vary from 5 to approximately 50 epochs. In other cases, a larger number of epochs may be required. In our problem statement, we assume continuous learning with different datasets. This means that each dataset can potentially employ a low number of epochs. On a long enough timeline, the

accuracy of each ML model will converge to the required. We assume that the number of epochs can be [1, 50, 100, 150, 250].

Table 2: GPU Inference Performance

Batch Size	Throughput in samples/sec
1	98
2	167
4	214
8	259
16	350
32	407

The running time of the ILP algorithm for the aforementioned parameters was approximately 0.7 seconds to create the equations and 0.8 seconds to prepare the solver (IBM CPLEX) and find the solution (the total number of variables was approximately 11000). The optimality gap was always 0.00%. For a larger instance of 60 edge nodes and 200 jobs the running time in total was approximately 2 and 2.5 seconds respectively (61000 variables). Therefore, the ILP algorithm can provide a timely solution even in large instances of the problem, and a heuristic is not necessary. Future variations of the problem can be more computationally intensive, and a heuristic could be required.

Table 3: Important Simulation Parameters

Symbol	Value	Symbol	Value
N	20 nodes	P_o	30 sec
R_n^G	200 GPUs	$\|T_{jx}\|$	2, 3, 4, 5 tasks
R_n^B	10 Tbps	$\|T_{ji}\|$	1, 2, 3 tasks
J	100 jobs	Π	166 samples/sec
C_C^{bw}	$0.02/GB	C_C^G	$2.08/hour

Inference Parameters

In the inference scenario we assumed that all jobs did not have constraints on propagation delay. Therefore, no job is fixed to be served at the edge. This allows for clear evaluation of the tradeoffs between edge/cloud b/w and processing costs and how these affect the allocation of the inference jobs. We assumed that each job consists of either 1, 2 or 3 tasks since typically, an inference job requires little amount of parallelism. The duration of the period is set at $P_o = 1$ second. One parameter that differentiates the evaluation of training from inference, is that the GPU performance of the inference varies depending on the batch size [19]. This means that depending on the number of samples that each task has to process, the tradeoffs between processing and b/w costs may differ. Thus, we assumed different $\sum_{e \in E_{jk}} \lambda_{e_{jk}}$ for each of the tasks: [1, 2, 4, 8, 16, 32]. According to [19] the performance of the GPU (NVIDIA DGX-1 with 1xV100 16G) for the inference is that of Table 2. To find the required number of GPUs each task requires, we also used Eq. 10.

3.1.4.2 Simulation Results

In this section we present the simulation results. We will present first the results for the training case, and then for the inference.

Training

a) Edge vs Cloud allocation decisions

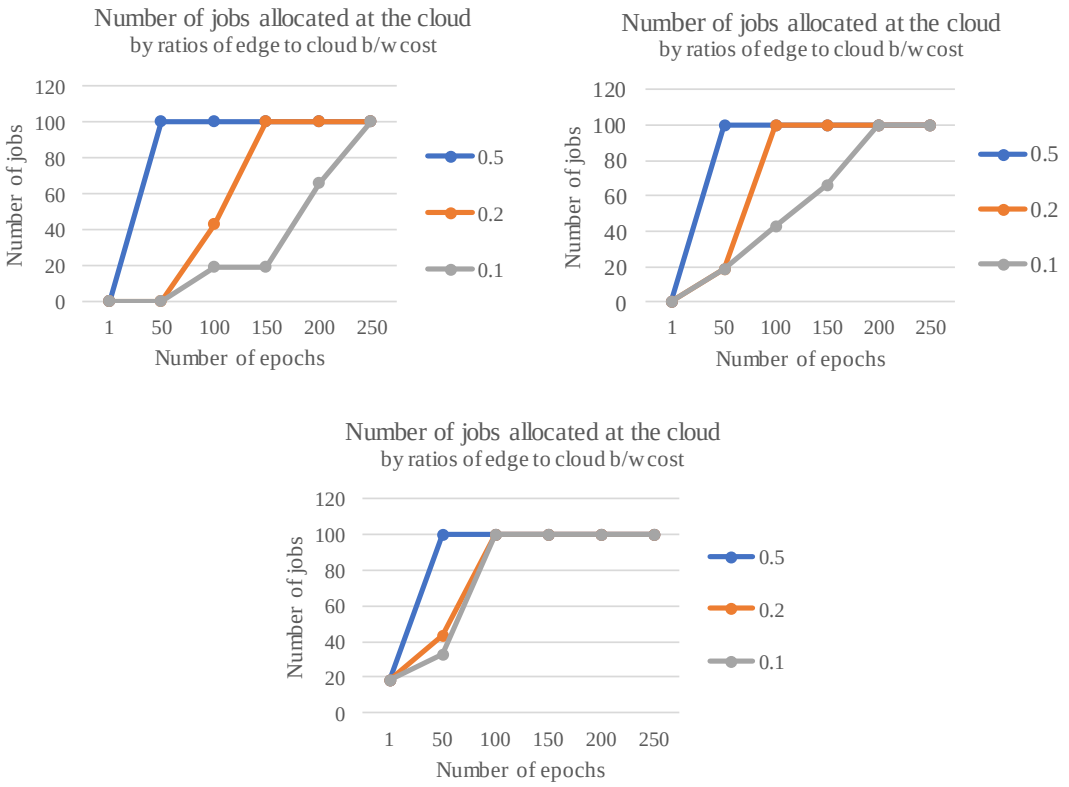

Figure 5: Number of jobs allocated at the cloud for different b/w costs and ratios of edge to cloud (a,b,c)

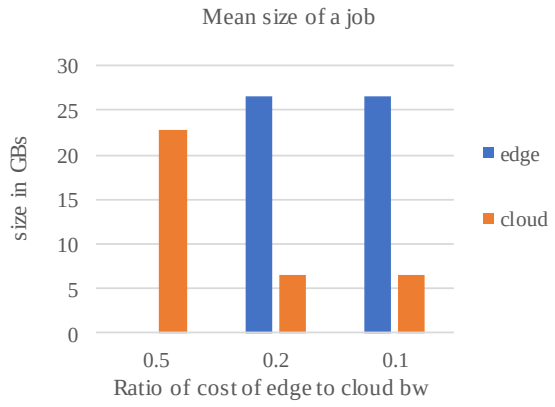

Figure 6: Mean size of a job allocated in edge or cloud for 50 epochs and for different cost ratios of edge to cloud b/w

In this subsection we will demonstrate how the different values of the assumed parameters affect the allocation of a training job at the edge or at the cloud. In Figure 5 we show the number of training jobs allocated at the cloud as a function of the number of epochs and for different edge/cloud processing and bandwidth (b/w) costs. For simplicity reasons we do not depict the allocation of the remaining jobs at the edge, since those are the remaining out of the 100 assumed. In Figure 5a the processing cost ratio of edge to cloud is 1.2. When the number of epochs is small, all jobs are served at the edge, since the b/w costs are lower. As the number of epochs increases, some jobs are served at the cloud depending on the b/w cost ratio. The increased number of epochs means that the total processing cost of a job play a more important role than the b/w cost to the allocation of the jobs. Even though the difference of edge and cloud processing costs is small, for large number of epochs the processing costs are so much greater than the bandwidth costs, that most or all jobs are served at the cloud. Note that as we will see in Figure 6, the cloud serves smaller (in terms of Mbytes) jobs. In Figure 5b, Figure 5c the edge processing costs are even more expensive than the cloud's. In Figure 5b we notice that the allocation of jobs tips towards the cloud relatively quickly. The different b/w costs still seem to play a role for the allocation of the jobs but their role is not significant. In Figure 5c more jobs are served at the cloud. Even for small number of epochs most jobs are always served at the cloud. Despite the edge processing costs being twice the cloud's, the edge is still more preferable until 50 epochs. Overall, from Figure 5we can conclude that the edge is more preferable to serve jobs with relatively low processing requirements. Also the different b/w cost ratios generally play a relatively small role in the allocation. A prominent difference is between ratio 0.5 and 0.1. In Figure 5b for example, the cloud can serve approximately two times more jobs when the ratio is 0.5 as opposed to 0.1 for large number of epochs.

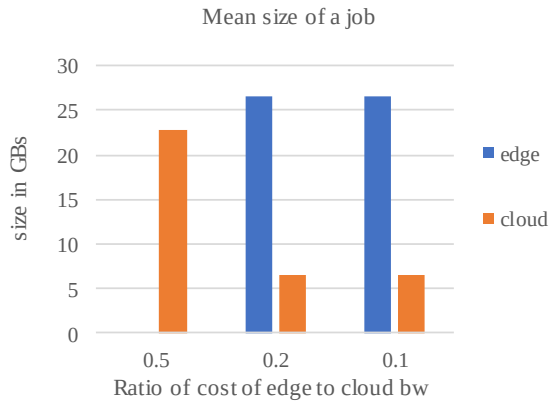

Figure 6 depicts the mean size in GBs for 40 epochs of a job's task that is served at either the edge or cloud when the edge's processing costs are twice than the cloud's. Similar conclusions can be drawn for different epochs and processing costs (as long as some jobs are served at the edge and others at the cloud). The size of a task depends on the number of samples/sec λ_u of its related devices, the duration of P_o, and the size of each sample of a task. The first two variables are the same for all the jobs we considered. Thus, the differentiating factor is the size of a task's sample. Note also that we have assumed a random number of tasks per job. This means that the definite size of a job depends also on

the exact number of the tasks. However, this does not significantly affect the decision on the allocated location of a job. The increased number of tasks not only means more data to transfer (hence increased b/w costs), but also means more samples to calculate (hence analogous increase in the processing requirements). Since we have assumed that the performance of a GPU in samples/sec is constant regardless of the size of a sample, the differentiating factor in whether a job will be served at the edge or at the cloud is the size of its tasks' samples. We notice that the edge tends to serve tasks with large size. It seems that in order for a task to be served at the cloud, it has to be significantly smaller than the tasks that are typically served at the edge. As the b/w cost decreases, the size of the jobs served to the cloud decreases respectively by 34%. The trends are similar for different edge processing costs. Also, when the number of epochs increases, the contribution of the processing costs due to the additional epochs is increased. Therefore, a job has to be larger to be served at the edge. Note also, that we considered an image recognition training scenario. Thus, each sample is relatively large. In different applications the size of the samples can be smaller. This means less b/w required, thus different job distribution at the edge and cloud. For example, we also evaluated Automated Speech Recognition training. In this scenario, the performance of the GPU is around 600 sequences/sec, while the size of a sample (word) is very small. Thus, the b/w costs are insignificant. In this scenario all jobs were served at the cloud when the edge processing costs were greater than the cloud's regardless of any other parameter.

b) Monetary cost evaluation

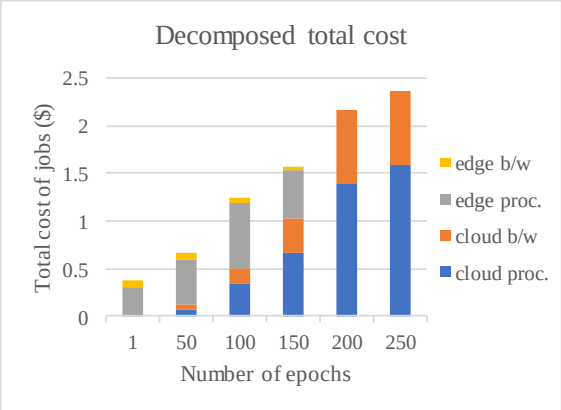

Figure 7: Total decomposed cost of jobs for edge/cloud cost b/w ratio 0.1 and processing ratio 1.5

In this subsection we will show how the different assumed parameters affect the related monetary costs of the jobs served at the edge and at the cloud. Figure 7 shows the total cost to serve all the jobs for one training round, decomposed to edge and cloud b/w and processing costs and for different number of epochs. In this case we assumed an edge to cloud b/w cost ratio of 0.1, and edge to cloud processing costs of 1.5. The main cost contributor is the processing. As the number of epochs increases, the edge (and later cloud) processing costs play the most important role in the total cost of the jobs. After 200 epochs all jobs are served at the cloud. Overall, we notice that the *cloud* b/w costs are a considerable fraction (approximately 35% for 200 and 250 epochs) of the overall cloud costs.

Thus, in lower number of epochs the edge is more preferable. Also, the (inexpensive) edge b/w costs are actually a sizeable part of the total edge costs for 1 and 50 epochs (20%, 13% respectively)

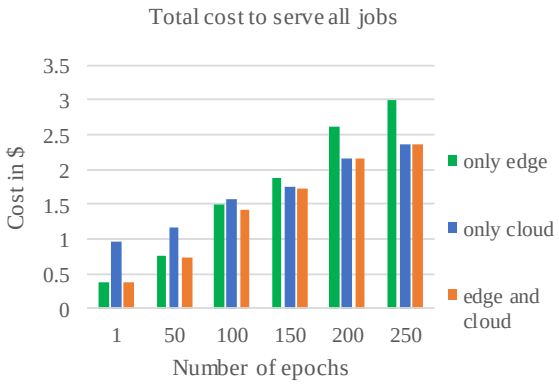

Figure 8: Total cost to serve the jobs for different number of epochs and for edge/cloud cost b/w ratio 0.1 and processing ratio 1.5

In Figure 8 we present the total cost to serve all the training jobs to the edge/cloud for the same parameters as Figure 7. We compare to the cost in case we had only cloud or only edge resources available. We notice that for low number of epochs the cloud is overall much more expensive (more than two times) than the edge. For 100 and 150 epochs the cooperation of edge and cloud is 4.8% and 7.5% less expensive than the edge, and 8.8%, 2% less expensive than the cloud respectively. For large training instances, this difference can be monetary significant. After 200 epochs all jobs are served to the cloud since the edge is more expensive overall. In either case, the cooperation of edge/cloud can result in significant monetary cost savings when compared to the isolated operation of either the edge or cloud.

Inference

a) Edge vs Cloud allocation decisions

In Figure 9 we show the number of inference jobs allocated at the cloud as a function of the number of samples per task and for different edge/cloud processing and b/w costs. In all (Fig. 7 a,b and c) figures, we notice that for a small number of samples per task most or all jobs are served at the cloud. As the number of task samples increases, so does the number of jobs served at the edge. The reason is that for the smaller number of task samples, the GPU performance (in samples/sec) is lower. Thus, the processing costs (for a given time period) are more significant than the respective b/w costs. As the number of task samples increases, so does the performance of the GPU and therefore the processing costs are reduced. Thus, more jobs are served at the edge. As expected, for larger processing edge/cloud cost ratios, more jobs are served at the cloud. The difference is significant for large number of samples per task (e.g., 100% more jobs are served at the cloud). Generally, the different b/w cost ratios do not play a big role in the allocation of jobs in the case of inference. However, the difference between the b/w edge/cloud cost ratio 0.5 and 0.1 is prominent after 4 samples per task. The cloud can serve approximately two times more jobs when the ratio is 0.5 compared to 0.1.

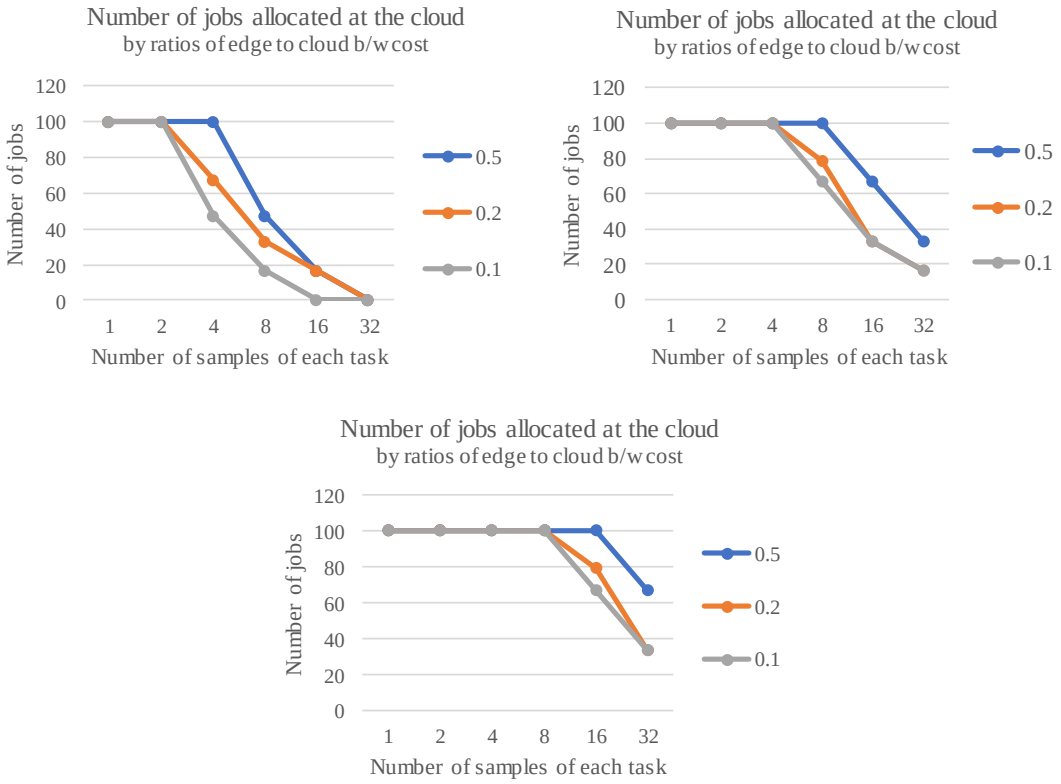

Figure 9: Number of inference jobs allocated at the cloud for different b/w costs and ratios of edge to cloud processing costs: (a) 1.2, (b) 1.5, (c) 2

b) Monetary cost evaluation

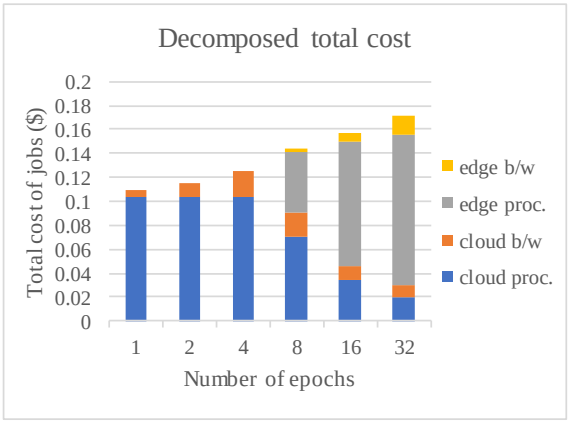

Figure 10: Total decomposed cost of jobs for edge/cloud cost b/w ratio 0.1 and processing ratio 1.5

Figure 10 shows the total cost to serve all the inference jobs for one round, decomposed to edge and cloud b/w and processing costs and for different number of samples per task. In this case we assumed an edge to cloud b/w cost ratio of 0.1, and edge to cloud processing costs of 1.5. As with the case of training, the contribution of the overall processing costs to the overall cost of the inference jobs is greater than that of overall b/w costs. A general trend for both cloud and edge b/w cost is that as the number of samples of each task increases, so does the contribution of the b/w costs to the total cost. More specifically, the cloud b/w cost starts from 4.8% and goes up to 22.4% and 31.5% of the overall *cloud* costs

for 1, 8 and 32 samples for each task respectively. Similarly, the edge b/w cost is starts from 4.1%, and goes up to 11.2% of the overall *edge* costs for 8 and 32 samples for each task respectively. As we mentioned in Figure 9 this is because the GPU has increased performance as the batch size increases.

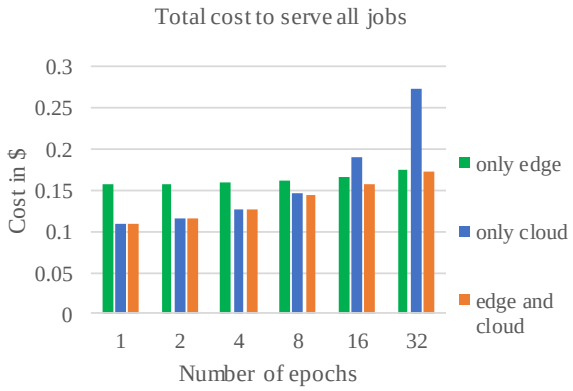

Figure 11: Total cost to serve the jobs for different number of epochs and for edge/cloud cost b/w ratio 0.1 and processing ratio 1.5

In Figure 11 we present the total cost to serve all the inference jobs to the edge/cloud for the same parameters as Figure 10. We compare to the cost in case we had only cloud or only edge resources available. The cooperation of edge and cloud provides benefits in the cases where the number of samples of a task is over 8. In the rest of the cases all the jobs are served at the cloud (see Figure 9c). The cooperation of edge/cloud is 10% and 2% less expensive than the exclusive use of the edge and the cloud respectively for the case of 8 samples for each task. For the case of 32 samples/task, the cooperation is 4.2% less expensive than the exclusive use of edge, and 16.4% less expensive than the exclusive use of cloud resources. The benefits of the cooperation can be greater for different b/w cost ratios (0.5 or 0.25). In these cases, there is a larger variety of allocation of jobs at the edge and at the cloud. Therefore, there are more opportunities for cost reduction.

3.1.5 Conclusions

In this work we considered the resource allocation problem for distributed ML training and inference applications. We proposed a framework to allocate resources for ML jobs at the edge–cloud continuum. The objective of the allocation is to minimize the monetary cost to serve the jobs, while respecting possible stringent timing constraints. We examined various optimization parameters pertained to processing costs and bandwidth costs in both edge and cloud resources. The results indicate that the processing costs play an important role in the allocation of a job at the edge or at the cloud. Significant cost savings were observed through the cooperation of edge and cloud resources when compared to the exclusive use of edge or cloud. Future work includes the possibility that an ML job (either training or inference) can be served (partially) at the device where the data are produced. Also, future work includes the modeling of energy consumption as well as prediction of the future workload to better manage the available network resources.

3.2 Secure Distributed Storage for the Cloud-Edge Infrastructure

Distributed storage systems store data in multiple cloud data centres and consolidate resources from multiple cloud providers. This can lead to increased flexibility as opposed to single storage services [26]. However, distributed storage systems struggle to address the heterogeneous and strict requirements of bandwidth-intensive and latency-sensitive applications, as the data are usually stored in different and probably distant locations. At the same time, edge computing brings the storage and processing of the data much closer to the generating source (e.g., camera or other sensors) [27]. The incorporation of edge resources, when properly used by the distributed storage services, can improve the way the demanding applications are served: a part of the data depending on the application can be stored and processed at the edge to minimize the latency and the network usage and if additional resources are required, then the abundant resources of the cloud can be utilized. In such a distributed model, the increase in the number and the density of the edge resources will inevitably change the operation of the current storage services. Next, we propose storage allocation mechanisms that leverage the erasure coding technique so as to store the data in a secured manner and consider a number of different optimization criteria, which are examined jointly and separately. Early results of this work were published and presented in IEEE CloudNet 2021 [62].

3.2.1 Description of the Infrastructure

The infrastructure includes three main components: the on-premise component (OPC), the cloud nodes and the edge nodes. The OPCs are privately owned devices which act as gateways between the user and the storage service. They perform critical operations such as files encryption, decryption along with the splitting and merging (based on the selected erasure code policy). The monetary costs for file hosting on either a cloud or an edge node and the average latency for data placement are assumed to be given by their corresponding providers. In general, the cloud nodes are located far from the OPCs, but their monetary charges are commonly lower than an edge node. On the other hand, the edge nodes are located closer to the OPCs, but their monetary charges are commonly higher than the charges of a cloud node. Finally, the cloud nodes provide much higher availability the edge ones, along with a nearly infinite amount of storage space.

3.2.2 Distributed Storage Operations

The operations that are performed in a distributed storage system are divided into (i) Data processing, (ii) Store and (iii) Retrieve operations. Each one of these operations depends on the number of the fragments k a file d is split into and the number of redundant fragments m, whose number depends on the selected erasure coding scheme and the considered optimization criteria. Other parameters that can affect the cost are (i) the file's size M_d (measured in Data Units - DU), (ii) the hosting duration T_d (measured in Period Units

- PU) and (iii) the number of expected retrievals τ_d within the hosting duration, which introduces a monetary and a latency cost.

3.2.2.1 Store/Retrieve Data Processing

A file d that is going to be hosted, initially, it is transferred to an on-premise component (OPC) ω_d via a secure connection. At the OPC, the file is split into k fragments of size M_d/k. These fragments are encoded given a $(k+m, m)$ erasure coding scheme and a total number of $k+m$ fragments are created. Note that in this way more fragments than the minimum required ones in order to retrieve a file are stored and this selection is considered during the optimization. The fragments in which the file is finally split into are encrypted and stored in the infrastructure. The splitting and encoding/decoding operations introduce a delay, which is denoted as $D_{spl\omega_d}$ and $D_{enc\omega_d}$. These delays are measured in Time Units (TUs) and are accumulated in the overall latency. For the store operation the introduced latency is:

$$g_{kmd} = \frac{k+m}{k} M_d \, D_{enc\omega_d} + k \, D_{spl\omega_d} \qquad (11)$$

To retrieve a file, the decryption and decoding of the fragments is performed at the OPC ω_d, which performs the linear combination of a subset of k fragments among all the $k+m$fragments. The decoded fragments are then merged into file d. The latency of decoding/decryption, denoted by $D_{dec\omega_d}$ (in TUs) per DU, is proportional to M_d, while the latency of merging, denoted by $D_{mer\omega_d}$ (in TUs/merge) per DU, is proportional to k. Therefore, the overall latency of the retrieve operation is derived as:

$$g'_{kmd} = M_d \, D_{dec\omega_d} + k \, D_{mer\omega_d} \qquad (12)$$

As a different number of fragments and erasure codes of different capabilities can be selected, an interesting trade-off exists between the number of fragments and the associated overhead, which can be exploited by the storage allocation mechanism we have developed towards further improving the performance of the infrastructure.

3.2.2.2 Store Operation

3.2.2.2.1 Store Operation Monetary Cost

Data fragments are stored in the available edge/cloud infrastructure at various locations. We denote by $\mathcal{N}_s$ the selected locations that host these fragments, with $|\mathcal{N}_s| = k+m$. Since each storage location $n \in \mathcal{N}_s$ charges P_{sn} Cost Units (CUs) for each DU, the monetary cost for storing the fragments of file d can be defined as follows:

$$\varphi_1(k, m, d) = \sum_{n \in \mathcal{N}_s} \frac{M_d}{k} T_d P_{sn} \qquad (13)$$

3.2.2.2.2 Store Operation Latency

The latency of the store operation depends on the: (i) processing latency, (ii) latency of placing the fragments into the storage locations, (iii) propagation latency, and (iv) transmission latency. To process each fragment, g_{kmd} TUs are required, while the latency

for placing the fragments into the storage locations requires D_{sn} TUs per DU. A transmission latency of $D_{bn\omega_d}$ per DU and a propagation delay of $D_{ln\omega_d}$ are also introduced, which depend on the network's link capacity and the distance between the OPC ω_d and the storage location respectively. Given the locations $\boldsymbol{\mathcal{N}}_s$ where data fragments are stored, the total latency for storing a file d is calculated as:

$$\varphi_2(k,m,d) = g_{kmd} + \sum_{n\in\boldsymbol{\mathcal{N}}_s} \frac{M_d}{k} D_{bn\omega_d} + \max_{n\in\boldsymbol{\mathcal{N}}_s}\left\{\frac{M_d}{k} D_{sn} + D_{ln\omega_d}\right\} \qquad (14)$$

3.2.2.3 Retrieve Operation

3.2.2.3.1 Retrieve Operation Monetary Cost

The retrieval cost is related to the subset $\boldsymbol{\mathcal{N}}_r \subseteq \boldsymbol{\mathcal{N}}_s$ ($|\boldsymbol{\mathcal{N}}_r| = k$) of nodes from which the fragments are retrieved. Given that each storage location $n \in \boldsymbol{\mathcal{N}}_r$ charges according to the number of GET requests that are required to retrieve the whole fragment, each GET request retrieves ρ DUs and costs P_{rn} CUs. Therefore, the monetary cost for the retrieve operation is calculated as:

$$\varphi_3(k,m,d) = \sum_{n\in\boldsymbol{\mathcal{N}}_r} \frac{1}{\rho}\frac{M_d}{k} P_{rn} \qquad (5)$$

3.2.2.3.2 Retrieve Operation Latency

Similarly, to the store operation latency, the retrieve operation introduces latency that consists of the: (i) processing latency, (ii) latency for recovering the fragments from the storage locations, (iii) propagation latency, and (iv) transmission latency. Particularly, the latency for recovering the k fragments from the storage locations requires D_{rn} TUs per DU. Next, the transmission latency $D'_{bn\omega_d}$ per DU and the propagation delay $D_{ln\omega_d}$ depend on the network's characteristics and the distance between the storage locations and the OPC ω_d. Finally, a processing latency of g'_{kmd} (in TUs) is required at the OPC ω_d for decrypting and decoding the k fragments and then merging them into file d. Assuming that the storage allocation mechanism selects the locations of $\boldsymbol{\mathcal{N}}_r$ for retrieving the datasets, then the latency for the retrieve operation is calculated as follows:

$$\varphi_4(k,m,d) = g'_{kmd} + \sum_{n\in\boldsymbol{\mathcal{N}}_r} \frac{M_d}{k} D'_{bn\omega_d} + \max_{n\in\boldsymbol{\mathcal{N}}_r}\left\{\frac{M_d}{k} D_{rn} + D_{ln\omega_d}\right\} \qquad (6)$$

3.2.3 Distributed Storage Resource Allocation

Given a distributed storage infrastructure, the storage resource allocation problem decides the way a set of files $\boldsymbol{\mathcal{D}}$ is served. Each file $d \in \boldsymbol{\mathcal{D}}$ is described by the tuple $(M_d, T_d, \tau_d, \omega_d, \boldsymbol{\mathcal{Q}}_d, \boldsymbol{\mathcal{K}}_d, \boldsymbol{\mathcal{M}}_d)$, where T_d is the hosting duration (in PUs), τ_d is the number of

future retrievals (offline problem), ω_d is the OPC where d is going to be processed at and $\boldsymbol{\mathcal{Q}}_d$ is the set of QoS requirements. Each file is initially processed at ω_d and it is split into fragments whose number lies in the set $\boldsymbol{\mathcal{K}}_d$. Next, a set of different erasure codes is used to encode the fragments. The choices of the number of fragments used for redundancy is denoted as $\boldsymbol{\mathcal{M}}_d$. A total of $k+m$ ($k \in \boldsymbol{\mathcal{K}}_d, m \in \boldsymbol{\mathcal{M}}_d$) fragments are produced.
The resource allocation mechanism decisions consist of the (i) number of fragments that each file is split into, (ii) the appropriate erasure code and (iii) the set of storage locations where the fragments are hosted on, so as to minimize the weighted cost that results from the different optimization criteria.

3.2.3.1 Pre-processing Phase – Availability

To consider the availability of the file d, we make use of a pre-processing phase in which we calculate the combinations of $k+m$ storage nodes that can be used to host the fragments, $\forall k \in \boldsymbol{\mathcal{K}}_d$ and $\forall m \in \boldsymbol{\mathcal{M}}_d$. All fragments are stored over the infrastructure under the assumption that each storage node hosts at most one fragment of each file. Each node $n \in \boldsymbol{\mathcal{N}}$ is characterized by an availability factor A_n. There are $\Phi_{km} = \frac{|\boldsymbol{\mathcal{N}}|}{k+m}$ combinations for choosing $k+m$ nodes, denoted as $\boldsymbol{\mathcal{J}}_{1km}, \boldsymbol{\mathcal{J}}_{1km}, \cdots, \boldsymbol{\mathcal{J}}_{\Phi_{km}}$. Since $0 \sim m$ node failures can be tolerated, the availability is calculated by summing the probabilities of $\kappa \in [k, k+m]$ nodes being available. We denote the number of collections of κ available nodes as $\Theta = \frac{|\boldsymbol{\mathcal{J}}_{ikm}|}{\kappa}$ and the j-th collection as ${S_j}^{\Theta}$. The availability of a file that is hosted by the storage nodes in $\boldsymbol{\mathcal{J}}_{ikm}$ is derived as:

$$a_{ikm} = \sum_{\kappa=k}^{k+m} \sum_{j=1}^{\Theta} \left[\prod_{n \in {S_j}^{\Theta}} A_n \prod_{n \in \boldsymbol{\mathcal{J}}_{ikm} \setminus {S_j}^{\Theta}} (1 - A_n) \right] \qquad (7)$$

where $\boldsymbol{\mathcal{J}}_{ikm} \setminus {S_j}^{\Theta}$ denotes the unavailable nodes. Next, we prune the combinations that do not meet the minimum availability requirement $A_{req} \in \boldsymbol{\mathcal{Q}}_d$. We denote the indices of the combinations that meet the requirement as $\boldsymbol{\mathcal{I}}_{km} = \{i \mid a_{ikm} \geq A_{req} \in \boldsymbol{\mathcal{Q}}_d$ and $i = 1,2,\cdots,\Phi_{km}\}$, $\forall k \in \boldsymbol{\mathcal{K}}_d$ and $\forall m \in \boldsymbol{\mathcal{M}}_d$. Hence, the availability objective is:

$$\phi_5(k,m,d) = \max_{i \in I_{km}} \{a_{ikm} \mid k \in \boldsymbol{\mathcal{K}}_d \wedge m \in \boldsymbol{\mathcal{M}}_d\} \qquad (8)$$

3.2.3.2 Stochastic availability

In our approach, we study the offline version of the problem. Therefore, according to the availabilities A_n of the storage nodes, we compute a stochastic input for whether each node is available during each retrieve operation or not. This is basically a Bernoulli process. We consider a set of Bernoulli random variables $Y_{ndt} \sim \text{Bernoulli}(A_n)$, $n \in \boldsymbol{\mathcal{D}}, d \in \boldsymbol{\mathcal{D}}, t = 1,2,\ldots,\tau_d$. Therefore, the input is computed as:

$$H(n,d,t) = \begin{cases} 1, & \text{if } Y_{ndt} = 1 \\ 0, & \text{else} \end{cases}$$

3.2.3.3 Mixed-Integer Linear Programming Formulation

In this section we present the Mixed-Integer Linear Programming (MILP) formulation of the distributed storage resource allocation mechanism.

Table 4: MILP variable description

Variable	Description
v_{nd}	Binary variable that indicates whether a fragment of file $d \in \mathcal{D}$ is placed at location $n \in \mathcal{N}$.
y_{kmd}	Binary variable that indicates the splitting configuration $k \in \mathcal{K}_d$ and the erasure code $(k+m, m)$, $m \in \mathcal{M}_d$.
x_{nkmd}	Binary variable that indicates whether a fragment of file $d \in \mathcal{D}$ is placed at location $n \in N$, the splitting configuration $k \in \mathcal{K}_d$ and the erasure code $(k+m, m)$, $m \in \mathcal{M}_d$, which is used for file d.
z_{nkmdt}	Binary variable that indicates whether the fragment of file $d \in \mathcal{D}$ which is placed at location $n \in \mathcal{N}$, is retrieved during the t^{th} $(t = 1,2,\cdots,\tau_d)$ retrieve operation, while the splitting configuration is $k \in \mathcal{K}_d$ and the erasure code is $(k+m, m)$, $m \in \mathcal{M}_d$.
ξ_{nd}	Binary variable that indicates the fragment and the storage node $n \in \mathcal{N}$ with the maximum propagation and placing delay in store operation, for file $d \in \mathcal{D}$.
ξ'_{ndt}	Binary variable that indicates the fragment and the storage node $n \in \mathcal{N}$ with the maximum propagation and recovery delay, in the t^{th} $(t = 1,2,\cdots,\tau_d)$ retrieve operation, for file $d \in \mathcal{D}$.
ψ_d	Integer variable, which denotes the maximum value of propagation and placing delay in store operation, for each file $d \in \mathcal{D}$.
ψ'_{dt}	Integer variable, which denotes the maximum value of propagation and recovery delay, in the t^{th} $(t = 1,2,\cdots,\tau_d)$ retrieve operation, for each file d.
ζ_{ikmd}	Binary variable that indicates the combination $i \in \mathcal{I}_{km}$ of storage locations, the splitting configuration $k \in \mathcal{K}_d$ and the erasure code $(k+m, m)$, $m \in \mathcal{M}_d$, used for each $d \in \mathcal{D}$.

C1 $$\sum_{k\in\mathcal{K}_d}\sum_{m\in\mathcal{M}_d}\frac{M_d}{k}D_{sn}x_{nkmd} + \sum_{k\in\mathcal{K}_d}\sum_{m\in\mathcal{M}_d}D_{ln\omega_d}x_{nkmd} - \psi_d \leq 0, \quad n\in\mathcal{N}, d\in\mathcal{D}$$

C2 $$-\sum_{k\in\mathcal{K}_d}\sum_{m\in\mathcal{M}_d}\frac{M_d}{k}D_{sn}x_{nkmd} - \sum_{k\in\mathcal{K}_d}\sum_{m\in\mathcal{M}_d}D_{ln\omega_d}x_{nkmd} - U_d\xi_{nd} + \psi_d \leq U_d, \quad n\in\mathcal{N}, d\in\mathcal{D}$$

C3 $$\sum_{n\in\mathcal{N}}\xi_{nd} = 1, d\in\mathcal{D}$$

C4 $$\sum_{k\in\mathcal{K}_d}\sum_{m\in\mathcal{M}_d}\frac{M_d}{k}D_{rn}z_{nkmdt} + \sum_{k\in\mathcal{K}_d}\sum_{m\in\mathcal{M}_d}D_{ln\omega_d}z_{nkmdt} - \psi'_{dt} \leq 0, \quad n\in\mathcal{N}, d\in\mathcal{D}, t=1,2,\cdots,\tau_d$$

C5 $$-\sum_{k\in\mathcal{K}_d}\sum_{m\in\mathcal{M}_d}\frac{M_d}{k}D_{rn}z_{nkmdt} - \sum_{k\in\mathcal{K}_d}\sum_{m\in\mathcal{M}_d}D_{ln\omega_d}z_{nkmdt} - U'_d\xi'_{ndt} + \psi'_{dt} \leq U'_d, \quad n\in\mathcal{N}, d\in\mathcal{D}, t=1,2,\cdots,\tau_d$$

C6 $\sum_{n\in\mathcal{N}} \xi'_{ndt} = 1, d \in \mathcal{D}, t = 1,2,\cdots,\tau_d$

C7 $\sum_{d\in\mathcal{D}} \sum_{k\in\mathcal{K}_d} \sum_{m\in\mathcal{M}_d} \frac{M_d}{k} x_{nkmd} \leq C_n, n \in \mathcal{N}$

C8 $\sum_{n\in\mathcal{N}} v_{nd} \leq \sum_{k\in\mathcal{K}_d} \sum_{m\in\mathcal{M}_d} (k+m) y_{kmd}, d \in \mathcal{D}$

C9 $\sum_{k\in\mathcal{K}_d} \sum_{m\in\mathcal{M}_d} y_{kmd} = 1, d \in \mathcal{D}$

C10 $\sum_{n\in\mathcal{N}} \sum_{k\in\mathcal{K}_d} \sum_{m\in\mathcal{M}_d} z_{nkmdt} \leq \sum_{k\in\mathcal{K}_d} \sum_{m\in\mathcal{M}_d} k y_{kmd}, d \in \mathcal{D}, t = 1,2,\cdots,\tau_d$

C11 $z_{nkmdt} \leq x_{nkmd}, n \in \mathcal{N}, d \in \mathcal{D}, k \in \mathcal{K}_d, m \in \mathcal{M}_d, t = 1,2,\cdots,\tau_d$

C12 $-x_{nkmd} + v_{nd} + y_{kmd} \leq 1, n \in \mathcal{N}, d \in \mathcal{D}, k \in \mathcal{K}_d, m \in \mathcal{M}_d$

C13 $x_{nkmd} - v_{nd} \leq 0, n \in \mathcal{N}, d \in \mathcal{D}, k \in \mathcal{K}_d, m \in \mathcal{M}_d$

C14 $x_{nkmd} - y_{kmd} \leq 0, n \in \mathcal{N}, d \in \mathcal{D}, k \in \mathcal{K}_d, m \in \mathcal{M}_d$

C15 $-\sum_{n\in\mathcal{N}} \theta(i,k,m,n) x_{nkmd} + U''_{ikm} \zeta_{ikmd} \leq U''_{ikm} - \sum_{n\in\mathcal{N}} \theta(i,k,m,n) x_{nkmd},$
$d \in \mathcal{D}, k \in \mathcal{K}_d, m \in \mathcal{M}_d, i \in \mathcal{I}_{km}$

C16 $\sum_{i\in\mathcal{I}_{km}} \zeta_{ikmd} = y_{km}, d \in \mathcal{D}, k \in \mathcal{K}_d, m \in \mathcal{M}_d$

C17 $\sum_{i=1}^{5} w_i = 1$

The examined objective includes the monetary cost for the store and retrieve operations, the latency and the average availability of the file. Constraints C1-C3 are used to calculate the latency when the fragments are stored at the different storage locations, while constraints C4-C6 are used for the calculation of the retrieval latency. Constraint C7 ensures that the placement of the fragments has been performed based on the available storage capacity and constraint C8 ensures that all fragments are hosted by the infrastructure. Constraint C9 ensures that a selection of the number of fragments and the erasure code is performed. Constraints C10-C14 express that the minimum number of fragments is retrieved along with the respective storage locations. Finally, constraints C15-C16 spot the locations where the fragments for a specific file are placed and C17 is a weighting coefficient related constraint.

3.2.3.4 Heuristic Algorithm

For large instances, the optimal MILP mechanism requires a high execution time. For this reason, we also developed a heuristic algorithm which relies on a suboptimal best policy. It serves the demands sequentially by creating a partial trajectory of the different file

fragmentation options. Then it executes a rollout-based algorithm [40] that operates as if each partial trajectory was the only one and selects for each file the resource allocation with the minimum cost. The pseudocode of the proposed mechanism is presented in Algorithm 1.

Algorithm 1

Input: $w, \boldsymbol{\mathcal{N}}, \boldsymbol{\mathcal{D}}, \rho, H, \{P_{sn}, P_{rn}, D_{sn}, D_{rn} \mid \forall n \in \boldsymbol{\mathcal{N}}\}, \{\boldsymbol{\mathcal{K}}_d, \boldsymbol{\mathcal{M}}_d, \boldsymbol{\mathcal{Q}}_d \mid \forall d \in \boldsymbol{\mathcal{D}}\}$
$\{D_{ln\omega_d}, D_{spl\omega_d}, D_{mer\omega_d}, D_{enc\omega_d}, D_{dec\omega_d} \mid \forall n \in \boldsymbol{\mathcal{N}}, \forall d \in \boldsymbol{\mathcal{D}}\}$
Output: $nodes(d), erasure\_code(d), \forall d \in \boldsymbol{\mathcal{D}}$ and $tcost$

for $d \in \boldsymbol{\mathcal{D}}, n \in \boldsymbol{\mathcal{N}}$ **do**
  $h\_cost(n,d) \leftarrow 0$
  for $t \in \{1, 2, \dots, \tau_d\}$ **do**
    $h\_cost(n,d) \leftarrow h\_cost(n,d) + H(n,d,t)$
  end for
end for
for $d \in \boldsymbol{\mathcal{D}}$ **do**
  $CM_d \leftarrow \{\}$
  for $k \in \boldsymbol{\mathcal{K}}_d, m \in \boldsymbol{\mathcal{M}}_d$ **do**
    $CS \leftarrow \{\}$
    for $n \in \boldsymbol{\mathcal{N}}$ **do**
      if $C_n \geq M_d$ **then**
        $par\_node(n) \leftarrow \{P_{sn}, P_{rn}, D_{sn}, D_{rn}, A_n\}$
        $cost(n,d) \leftarrow calc\_node\_cost(n,k,m,d,par\_node(n))$
        $CS \leftarrow CS \cup \{n\}$
      end if
    end for
    $CS^* \leftarrow$ Sort CS^* according to $h\_cost(n,d)$ and $cost(n,d)$
    $\boldsymbol{\mathcal{N}}^* \leftarrow \{CS^*[1], CS^*[2], \cdots, CS^*[k+m]\}$
    $CM_d \leftarrow CM_d \cup \{(k, m, \boldsymbol{\mathcal{N}}^*)\}$
  end for
end for
for $d \in \boldsymbol{\mathcal{D}}$ **do**
  $fcost_{\min} \leftarrow \infty$
  for $(k, m, \boldsymbol{\mathcal{N}}^*) \in CM_d$ **do**
    $par\_opc(n, \omega_d) \leftarrow \{D_{ln\omega_d}, D_{spl\omega_d}, D_{mer\omega_d}, D_{enc\omega_d}, D_{dec\omega_d}, par\_node(n)\}$
    If $calc\_file\_cost(w, d, \boldsymbol{\mathcal{N}}^*, par\_opc(n, \omega_d)) < fcost_{\min}$ **then**
      $fcost_{\min} \leftarrow calc\_file\_cost(w, d, \boldsymbol{\mathcal{N}}^*, par\_opc(n, \omega_d))$
      $CM_{\min} \leftarrow (k, m, \boldsymbol{\mathcal{N}}^*)$
    end if
  end for
  $(k, m, \boldsymbol{\mathcal{N}}^*) \leftarrow CM_{\min}$
  $nodes(d) \leftarrow \boldsymbol{\mathcal{N}}^*, erasure\_code(d) \leftarrow (k+m, m)$
  Update $C_n, \forall n \in \boldsymbol{\mathcal{N}}^*$; update $tcost$
end for

3.2.4 Simulation Experiments

3.2.4.1 Simulation Setup

In this section, we present the performance evaluation of the proposed MILP and heuristic mechanisms through simulations. The MILP mechanism is written in Python and the Gurobi Optimizer [41] is used. The experiments are performed on a computer with an Intel Core i7-9700K processor running at 3,6 GHz and 32 GB of RAM. We examined performance of the heuristic mechanism versus the optimal solution provided by the MILP mechanism for a set of different optimization criteria, namely (i) maximize the availability of the stored file, (ii) minimize the retrieve latency, (iii) minimize retrieve cost, (iv) maximize store latency, (v) maximize store cost, (vi) a weighted function that considers equally all the aforementioned criteria.
We have briefly described the components of the infrastructure in the Section 3.2.1. In this section we provide a more detailed description of the setup which we used in the simulations. We assumed two different cloud-edge storage infrastructures: a small one where the parameters are computed randomly and a large one where most of the parameters are provided from real-world data logs. We examined the offline scenario in which all the storage demands are known in advance.

There is only a single OPC in the small infrastructure, and the cloud and edge nodes are placed in random distances relatively to the OPC. Therefore, their monetary costs are also computed randomly according to the Table 5. Each file $d \in \boldsymbol{\mathcal{D}}$ has a size of $M_d = 5$ DUs, it is hosted for a period of $T_d = 10$ PUs and it is retrieved $\tau_d = 100$ times from the storage service. Furthermore, by default, all files are split into $k = 4$ fragments ($\boldsymbol{\mathcal{K}}_d = \{4\}, \forall d \in \boldsymbol{\mathcal{D}}$), which are encoded with an erasure code that introduces an overhead of $m = 2$ fragments ($\boldsymbol{\mathcal{M}}_d = \{2\}, \forall d \in \boldsymbol{\mathcal{D}}$). The minimum required availability for each file is set to 98%. Table 5 includes the monetary costs and latencies, the average availability A_n and the storage capacity C_n are presented in detail in the Table 5.

The large infrastructure corresponds to a real case scenario according to data exported by Chocolate Cloud's Skyflok[1]. We consider real-world coordinates and monetary charges for the cloud nodes. At each available city, we assume a randomly placed OPC that has access to all the cloud nodes. The edge nodes are placed randomly at each possible real-world country, relatively to a random OPC that resides in the same country. Their monetary charges are also random according to the Table 6. An OPC has access only to the edge nodes of the same country. The file sizes M_d, the hosting durations T_d and numbers of retrievals τ_d are based on the provided data by the CC. By default, all files are split into $k = 5$ fragments ($\boldsymbol{\mathcal{K}}_d = \{5\}, \forall d \in \boldsymbol{\mathcal{D}}$), which are encoded with an erasure code that introduces an overhead of $m = 4$ fragments ($\boldsymbol{\mathcal{M}}_d = \{4\}, \forall d \in \boldsymbol{\mathcal{D}}$). The default minimum required availability is set to 98%. Finally, the various components of the monetary costs and the latencies, the average availability A_n and the storage capacity C_n are presented in detail in the Table 6.

[1] SkyFlok is a multi-cloud distributed secure storage and sharing service developed by CC that provides file protection using encryption and novel erasure code (https://www.skyflok.com).

Table 5: Default parameters for the small infrastructure

Variable	Value		Unit
M_d	$5, \forall d \in \boldsymbol{\mathcal{D}}$		DUs
$\boldsymbol{\mathcal{K}}_d$	$\{4\}, \forall d \in \boldsymbol{\mathcal{D}}$		–
$\boldsymbol{\mathcal{M}}_d$	$\{2\}, \forall d \in \boldsymbol{\mathcal{D}}$		–
T_d	$10, \forall d \in \boldsymbol{\mathcal{D}}$		–
$\boldsymbol{\mathcal{Q}}_d$	$\{A_{req} = 98\%\}, \forall d \in \boldsymbol{\mathcal{D}}$		%
N_c	16		–
N_e	16		–
ρ	$10e^{-6}$		DUs per GET
$D_{spl\omega_d}$	$U(0.15,0.18)$		TUs per split
$D_{mer\omega_d}$	$U(0.12,0.144)$		TUs per merge
$D_{enc\omega_d}$	$U(300,360)$		TUs/DU
$D_{dec\omega_d}$	$U(150,180)$		TUs/DU
	Cloud nodes	**Edge nodes**	
D_{sn}	$U(300,360)$	$U(100,120)$	TUs/DU
D_{rn}	$U(150,180)$	$U(50,60)$	TUs/DU
$D_{ln\omega_d}$	$U(0.5,0.6)$	$U(0.05,0.06)$	TUs
P_{sn}	$U(0.25,0.310)$	$U(0.32,0.40)$	CUs/(DU*PU)
P_{rn}	$U(100e^{-8}, 120e^{-8})$	$U(130e^{-8}, 156e^{-8})$	CUs per GET
A_n	$U(99.9\%, 99.99\%)$	$U(70.0\%, 72.0\%)$	%
C_n	∞	$U(2500, 3250)$	DU

Table 6: Default parameters for the large infrastructure

Variable	Value		Unit
$\boldsymbol{\mathcal{K}}_d$	$\{5\}, \forall d \in \boldsymbol{\mathcal{D}}$		–
$\boldsymbol{\mathcal{M}}_d$	$\{4\}, \forall d \in \boldsymbol{\mathcal{D}}$		–
$\boldsymbol{\mathcal{Q}}_d$	$\{A_{req} = 98\%\}, \forall d \in \boldsymbol{\mathcal{D}}$		%
N_c	64 globally		–
N_e	16 per country		–
ρ	$10e^{-6}$		DUs per GET
$D_{spl\omega_d}$	$U(0.15,0.195)$		TUs per split
$D_{mer\omega_d}$	$U(0.12,0.156)$		TUs per merge
$D_{enc\omega_d}$	$U(300,390)$		TUs/DU
$D_{dec\omega_d}$	$U(150,195)$		TUs/DU
	Edge nodes		
D_{sn}	$U(100,120)$		TUs/DU
D_{rn}	$U(50,60)$		TUs/DU
$D_{ln\omega_d}$	$U(0.05,0.06)$		TUs
P_{sn}	$U(0.32,0.40)$		CUs/(DU*PU)
P_{rn}	$U(130e^{-8}, 156e^{-8})$		CUs per GET
	Cloud nodes	**Edge nodes**	
A_n	$U(99.9\%, 99.99\%)$	$U(70.0\%, 72.0\%)$	%
C_n	∞	$U(2500, 3250)$	DUs

3.2.4.2 Comparison between the heuristic and the MILP policy with random data

The two mechanisms are compared with respect to the monetary cost which is charged to the users for a number of files that varies in the close interval [20,100] (Figure 12 and Figure 13). The simulations are conducted with the data of Table 5 and Table 6.

As it is expected, in both cases, for all the considered optimization criteria, the total cost increases linearly according to the number of hosted files. In the case of the heuristic policy, when all optimization criteria are simultaneously taken into account the monetary cost ranges from 678 CUs with 20 hosted files to 3340 CUs with 100 hosted files indicating an average cost per file around 34 CUs. Additionally, when the objective is either the minimization of the store and retrieve operation latencies, the exhibited cost is higher than the case where all criteria are taken into account by 2.4% and 4.7% respectively. When the objective is either the minimization of the store or retrieve operation monetary cost, or the maximization of the availability, the exhibited cost is lower than the case where all criteria are taken into account, by 37.1%, 32.7% and 32.2% respectively.

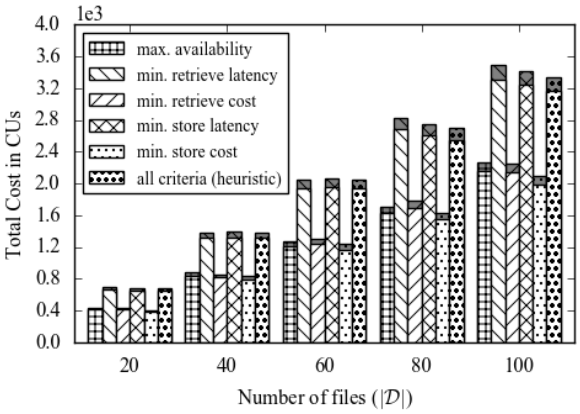

Figure 12: Evaluation of the number of files to the monetary costs, using the heuristic policy.

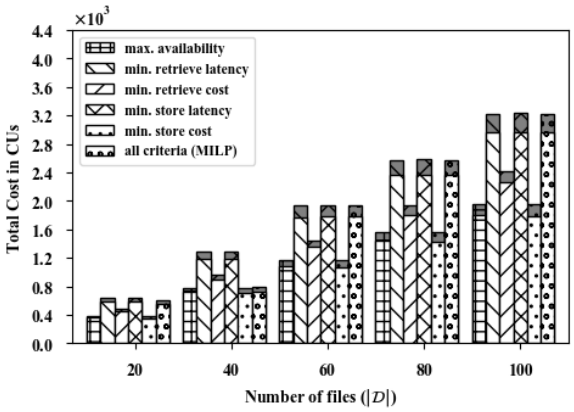

Figure 13: Evaluation of the number of files to the monetary costs, using the MILP and the heuristic policy.

In the case of the MILP policy when all optimization criteria are simultaneously taken into account the monetary cost ranges from 636 CUs with 20 hosted files to 3385 CUs with 100 hosted files indicating an average cost per file around 32 CUs. Additionally, when the objective is either the minimization of the store and retrieve operation latencies, the exhibited cost is about equal with the case where all criteria are taken into account. When the objective is either the minimization of the store or retrieve operation monetary cost, or the maximization of the availability, the exhibited cost is lower than the case where all criteria are considered, by 65.6%, 31.3% and 64.8% respectively.

Next, for the same optimization scenarios we examined the utilization of cloud and edge resources (Figure 14 and Figure 15) in Figure 14, when the objective is the minimization of the store or retrieve operation latency, the heuristic policy prefers the edge rather than the cloud resources. The results are similar in Figure 15 for the MILP mechanism. This is due to

the fact that the edge resources are much closer geographically to the end user than the locations of the cloud data centres.

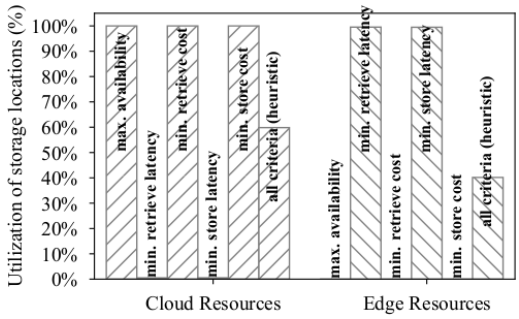

Figure 14: Effect of the optimization objectives to the percentage of utilized cloud and edge resources, using the heuristic policy with the large network

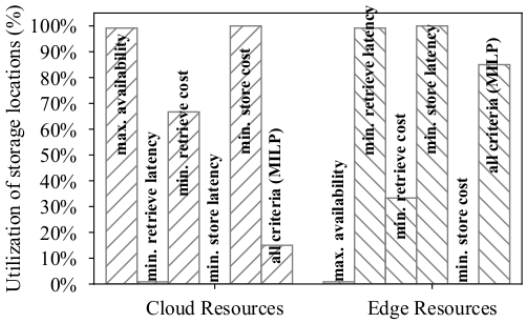

Figure 15: Effect of the optimization objectives to the percentage of utilized cloud and edge resources, using the MILP policy in the.

When the objective is either the minimization of the store or retrieve operation (monetary) cost, or the maximization of the availability of the files both the heuristic and the MILP mechanisms prefer to host the fragments on the cloud which is cheaper than the edge while it provides much higher availability. In particular, in all aforementioned cases, the heuristic policy utilizes only cloud resources (Figure 14), while the optimal solution provided by the MILP utilizes only cloud resources when it optimizes the monetary cost of the store operation and the availability but the cloud resource percentage reduces to 67% when the algorithm optimizes the retrieve operation cost.

Finally, when all criteria are optimized simultaneously, the heuristic policy prefers the edge instead of the cloud resources in 60.5% of the utilized storage nodes (Figure 14). Correspondingly, the MILP policy prefers the edge instead of the cloud resources in 83.3% of utilizing storage nodes (Figure 15). By comparing the figures, we can observe that the heuristic can achieve a solution that is really close to the solution of the optimal one regarding the utilization of the edge and cloud resources. The results are nearly identical in the four out of five individual criteria and are close enough when all criteria are considered simultaneously.

3.2.4.3 Effect of the number of files to the monetary cost of the service on real data

Having examined the optimality of the proposed heuristic mechanisms on the small cloud edge infrastructure, we continued our simulation experiments considering a real scenario described in Section 3.2.1 and we examined the effect of the optimization criteria into the monetary cost.

As it is shown in Figure 16, we present the total costs considering the whole real data logs which include the transaction entries and the characteristics of 12749 files, when the store and the retrieve operation latency is minimized, the average total cost per user is 22907 and 22735 CUs, respectively. On the other hand, when we minimize the store and the retrieve operation latency, the average total cost per user is 11827 and 12370 CUs, respectively.

When the availability of the files is targeted, the average total cost per user is 13518 CUs. Finally, when all the objectives are optimized simultaneously, the average total cost per user is 16228 CUs. Additionally, when the objective is either the minimization of the store and retrieve operation latencies, the exhibited cost is higher than the case where all the considered criteria are equally taken into account by 41.2% and 40% respectively. On the other hand, when the objective is either the minimization of the store or retrieve operation monetary cost or the maximization of the availability, the exhibited cost is lower than the case where all criteria are taken into account, by 27.1%, 23.8% and 16.7% respectively.

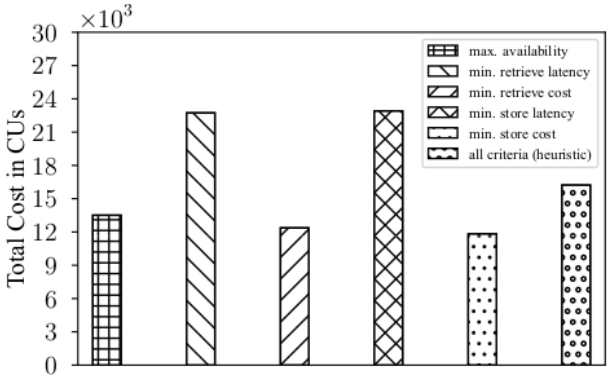

Figure 16: Evaluation of the monetary costs, using the heuristic policy.

Next, we examined the utilization of cloud and edge resources (Figure 17). We observed that, when the objective is the minimization of the store or retrieve operation latency, the heuristic policy utilizes only edge resources. On the other hand, when the objective is either the minimization of the store or retrieve operation (monetary) cost, or the maximization of the availability the heuristic policy prefers to host the fragments on the cloud which is cheaper than the edge while it provides much higher availability. Finally, when all criteria are optimized simultaneously, the heuristic policy prefers the cloud instead of the edge resources in 59.8% of the utilized storage nodes.

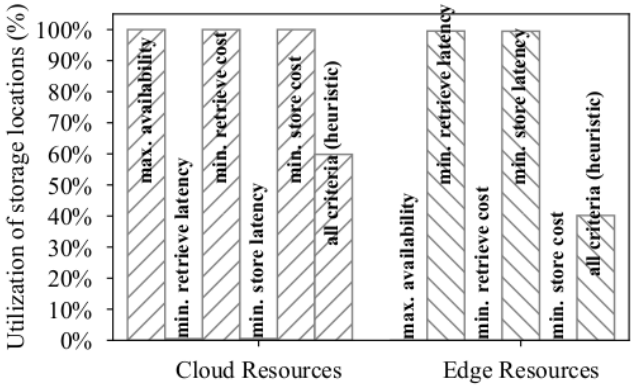

Figure 17: Effect of the optimization objectives to the percentage of utilized cloud and edge resources, using the heuristic policy.

3.2.4.4 Effect of the file fragmentation choices on the experienced latency on real data

In the following simulations, we examined the effect of the number of fragments each file is split into on the store and the retrieve operation latencies. In the first spart, we evaluate the heuristic policy for a network that deploys cloud storage nodes which are described in the real data logs and we examine various numbers of edge resources being deployed in the infrastructure. In our experiments, we deviate the numbers of fragments the files are split into, while the number of additional fragments for redundancy is fixed into four ($m = 4$).

In Figure 18 and Figure 20 we present the experienced store operation latency, while the retrieve operation latency is presented in Figure 19 and Figure 21. In Figure 18 and Figure 20 all optimization criteria are taken into consideration at the same time, while in Figure 19 and Figure 20 only the store or retrieve operation delay are optimized simultaneously while the monetary costs and the availability are not taken into consideration.

It is obvious that the number of fragments affects both the store and on the retrieve operation latencies. As the number of fragments increases, the store operation latency decreases while the retrieve operation latency decreases but when the number of fragments a file is split into more than $k = 10$ fragments it increases again. In particular, in the case of $(N_e, N_c) = (0{,}64)$ (Figure 18) the store operation latency steadily decreases from 5911 TUs with the erasure code of $(6{,}4)$ ($k = 2$) to around 2484 TUs with the erasure code of $(22, 4)$ ($k = 18$). Accordingly, there is a similar behavior in Figure 19 where the store operation latency decreases from around 5890 TUs with erasure code of $(6, 4)$ ($k = 2$) to 2469 TUs with the erasure code of $(22, 4)$ ($k = 18$). On the other hand, the retrieve operation latency in Figure 19 decreases from around 1258 TUs with erasure code of $(6, 4)$ ($k = 2$) to 1034 TUs with the erasure code of $(12, 4)$ ($k = 8$). From that point the latency gradually increases as the parameter k increases. Finally, in Figure 21 as in the previous cases, the retrieve operation latency decreases from 1235 TUs with erasure code of $(6, 4)$ ($k = 2$) to 1033 TUs with the erasure code of $(12, 4)$ ($k = 8$) and then it begins to increase. These cases exhibit maximal delays since only cloud resources are available.

This is explained considering that initially the splitting of the file to more fragments, leads to performing the data recovery from multiple providers in parallel, which speeds up the process. However, increasing the number of fragments even further leads to the addition of cloud resources that are less approximate to the on-premise component. Since the store or retrieve operation is considered to be complete only when the last fragment reaches its destination the delay increases.

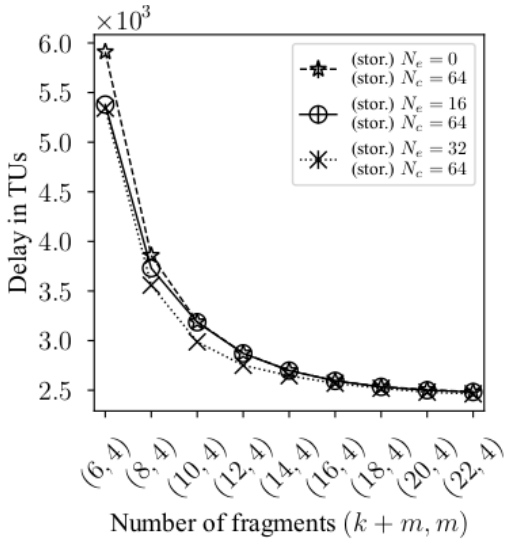

Figure 18: Effect of the number of fragments the files are split into, on store operation delay while optimizing all criteria simultaneously (heuristic policy).

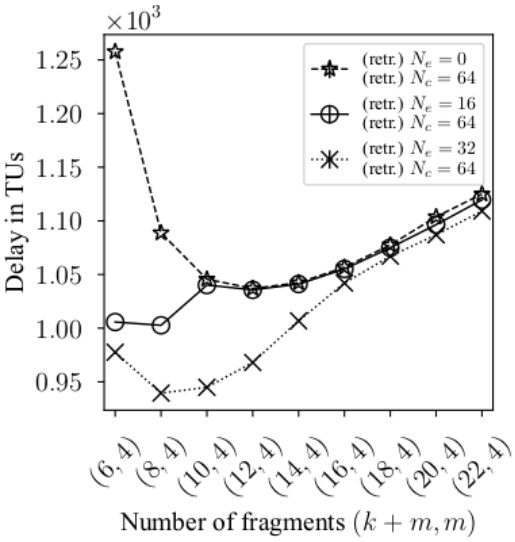

Figure 19: Effect of the number of fragments the files are split into, on retrieve operation latency while optimizing all criteria simultaneously (heuristic policy).

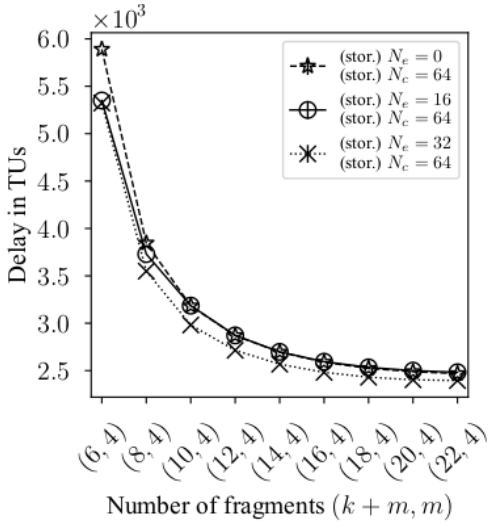

Figure 20: Effect of the number of fragments the files are split into, on store operation latency, while optimizing store and retrieve operation latencies (heuristic policy).

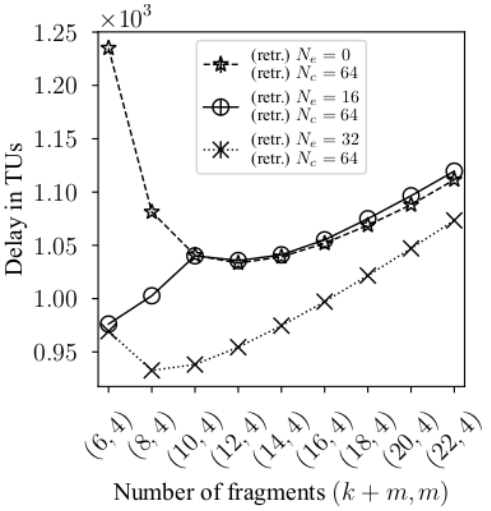

Figure 21: Effect of the number of fragments the files are split into, on retrieve operation latency, while optimizing store and retrieve operation latencies (heuristic policy).

In the case of $(N_e, N_c) = (16, 64)$ edge and cloud resources respectively, the selection of edge resources leads to a decrease in the delay, compared to the case where only cloud resources are considered. In particular, although the store operation latency initially is lower, when more than 10 fragments are selected the delay is identical to the previous case since the algorithm selects some cloud storage resources to host the fragments. This is depicted in a clearer way in Figure 20, where the objective is the minimization of latencies. In this case the delay reaches the levels of the previous case when the erasure code is $(18, 4)$ $(k = 12)$, which is the point where the edge resources are not sufficient to host all the fragments $(k + m = 18$ fragments are transmitted). The retrieve operation latency

(Figure 19) initially is low due to the selection of edge resources. However, from the erasure code of $(10, 4)$ and on, the algorithm performs similarly to the previous case due to the introduction of cloud resources to the solution. Again, this fact is clearer in Figure 21 where this sharp delay increase takes place with the erasure code of $(22, 4)$, that is where all required $k = 18$ fragments can't be retrieved from only 16 edge resources. Finally, in the case of $(N_e, N_c) = (32, 64)$ the edge resources are sufficient to store all the fragments, which leads to the lowest latencies, compared to the previous cases.

3.2.4.5 Effect of the file redundancy choices on the experienced latency on real data

In the following simulations, we examined the effect of the fragments on the store and the retrieve operation latency. We examine various numbers of edge resources being deployed in the infrastructure.

For our evaluations we considered the store operation latency in Figure 22 and Figure 24 and the retrieve operation latency in Figure 23 and Figure 25. While we deviate the numbers of fragments m used for redundancy, we keep the number of fragments the files are initially split into $k = 5$, for all our samples. In Figure 22 and Figure 23 all optimization criteria are taken into consideration at the same time, while in Figure 24 and Figure 25 only the store or retrieve operation delay are optimized simultaneously while the monetary costs and the availability are not taken into consideration.

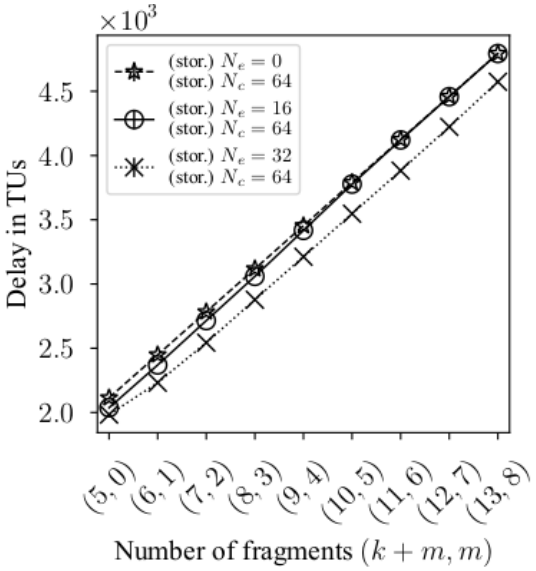

Figure 22: Effect of the number of fragments used for redundancy, on store operation delay while optimizing all criteria simultaneously (heuristic policy).

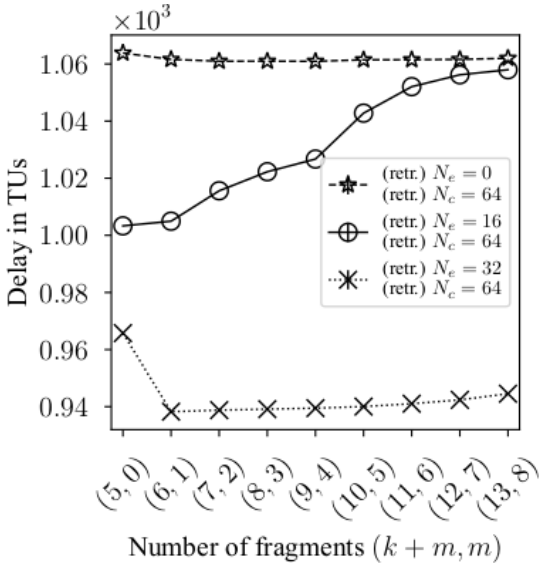

Figure 23: Effect of the number of fragments used for redundancy, on retrieve operation latency while optimizing all criteria simultaneously (heuristic policy).

As it is shown on the figures, increasing the redundancy leads to a linear increase of the store operation latency (Figure 22 and Figure 24). This is expected since the total size of the data increases linearly with the addition of a larger number of fragments. Again, the introduction of a larger number of edge resources to the infrastructure, decreases both the store and the retrieve operation. However, the retrieve operation latency. However, the retrieve operation exhibits a more complex behaviour.

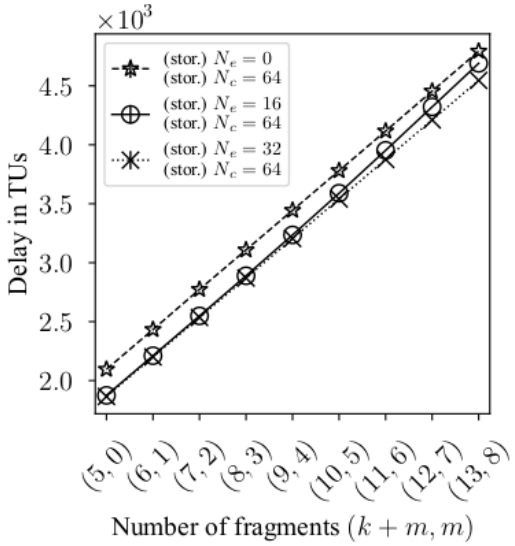

Figure 24: Effect of the number of fragments used for redundancy, on store operation latency, while optimizing store and retrieve operation latencies (heuristic policy).

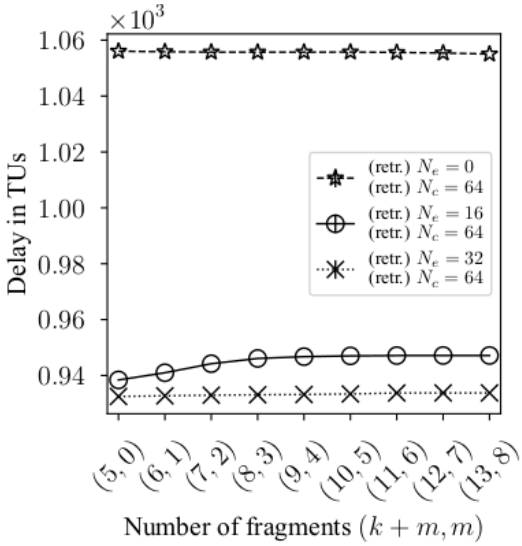

Figure 25: Effect of the number of fragments used for redundancy, on retrieve operation latency, while optimizing store and retrieve operation latencies (heuristic policy).

In Figure 23, in the case of $(N_e, N_c) = (0, 64)$ the algorithm achieves the highest average latency per retrieve operation by selecting only cloud resources, while in the case of $(N_e, N_c) = (32, 64)$ achieves the lowest average latency per retrieve operation by selecting only edge resources. This is also the case in Figure 25. The latency is not affected by the addition of redundancy, since the fragments are always retrieved from $k = 5$ out of the $k + m$ which are hosted by the infrastructure. In the mid-case of $(N_e, N_c) = (16, 64)$, the latency increases in both Figure 23 and Figure 25, but in the former case with a higher pace. In Figure 23, which depicts the multi-objective case, the algorithm selects to retrieve a number of fragments from cloud resources even though the 16 edge devices per country are sufficient to host $k = 5$ which are required. On the other hand, in Figure 25 only edge resources are selected to host the fragments and to recover the fragments from. However, the latency is higher compared to the case of $N_e = 32$, because having a lower number of available selections of edge resources increases the average maximum distance between the on-premise component and the selected resources.

All in all, increasing the redundancy greatly increases the store operation latency and to some variable level the retrieve operation latency. However, the retrieve operation latency increases the number and the density of the edge resources which are deployed in the infrastructure, providing more options.

3.2.4.6 Effect of the erasure code to the total monetary cost on real data

In Figure 26 we examined the effect of the erasure code selection on the monetary cost. We considered different splitting options for the files, $k = 2, 3, 4$ fragments, and used $m = 0, 1, 2$ additional fragments for redundancy, while all the optimization criteria were equally taken into consideration. In the first three samples, $(2, 0)$, $(3, 0)$ and $(4, 0)$ we did not use any fragments for redundancy. The files can be split into $k = 2, 3, 4$ fragments which are all required for the file recovery. In this case as the number of fragments increases the total cost slightly increases from 2213 to 2240 CUs as well. The algorithm utilizes storage nodes that provide an optimal cost. Therefore, as the number of storage nodes grows, the monetary charges of the additional storage nodes tend to become higher.

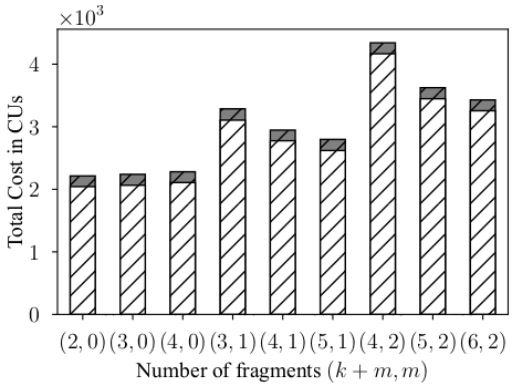

Figure 26: Effect of the erasure code policy to the total monetary costs.

In the next three samples, namely $(3, 1)$, $(4, 1)$ and $(5, 1)$, the files are split into $k = 2, 3$ and 4 fragments respectively, while an additional fragment is introduced for redundancy in all cases. The additional fragment has the same size as the rest of them which is M/k. Therefore, the total data size which is placed at the storage locations during the store operation is $M + M/k$ which is interpreted to $M + M/1$, $M + M/2$, $M + M/3$ for these particular samples. The total size is proportional to the store operation cost which explains the cost decrease.

Accordingly, in the last three samples, i.e. $(4, 2)$, $(5, 2)$ and $(6, 2)$, 2 additional fragments are used for redundancy with size M/k each. Therefore, the total file size is $M + 2 \cdot M/k$, which translates to $M + 2M/1$, $M + 2M/2$, $M + 2M/3$ for these particular samples.

The presented results confirm that the total monetary cost is proportional to the commodity $M + m \cdot M/k = (k + m) \cdot M/k$.

3.2.4.7 Effect of the minimum availability requirement to the selection of redundancy selection and to the type of storage resources on real data

In Figure 27 we examined the effect of the minimum availability requirement of the user to redundancy selection. All files were split into $k = 5$ fragments, while the number of fragments which are used for redundancy belongs to a set $m \in \{0, 1, 2, 3, 4, 5, 6\}$.

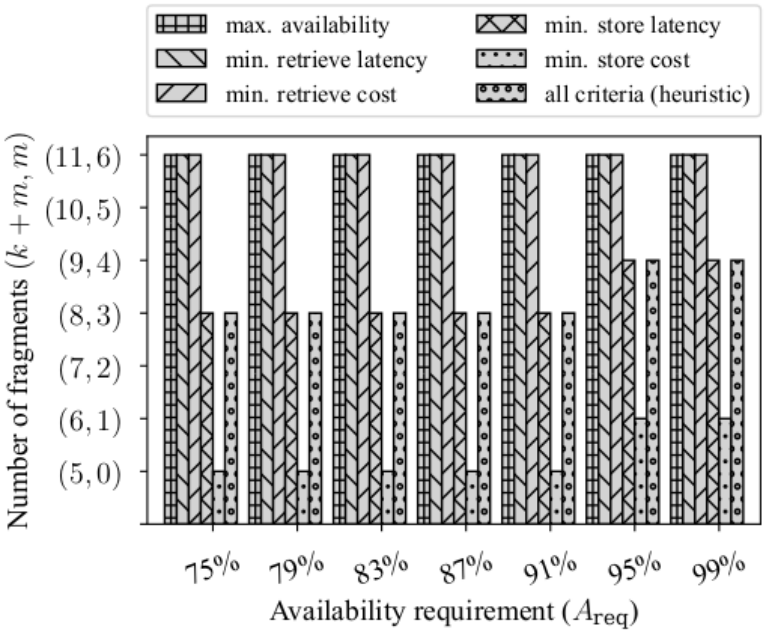

Figure 27: Effect of the minimum availability requirement to the selection of the level of redundancy.

We observe that when the availability is the only optimization criterion, 6 redundancy fragments are chosen, which is the maximum possible selection. In all other cases, the results exhibit an increase in the redundancy fragments as the requirement for availability increases. In general, when we minimize the store operation monetary cost, lower redundancy is preferred compared to the other cases. On the other hand, the retrieve operation cost is independent of the redundancy, since data of size M are retrieved each time. As a result, when the store operation cost is minimized, the algorithm minimizes the redundancy as well, as long as the constraint for the minimum availability holds. This translates to choosing $m = 0$ redundancy fragments when $A_{req} = 75\%, 79\%, 83\%, 87\%, 91\%$ and $m = 1$ when $A_{req} = 95\%, 99\%$. Accordingly, when the retrieve operation cost is minimized, redundancy is not considered. Therefore, the algorithm chooses any number of fragments, as long as the availability requirement is met.

In the case of minimizing the store operation latency, the algorithm tends to choose edge instead of cloud resources to host the data. Since the edge has lower average availability than the cloud, a larger number of resources is chosen for the availability constraint to be met. Additionally, the algorithm tends to minimize the number of redundancy fragments for the following reason: when the store operation takes place, all $k + m$ fragments are transmitted to their corresponding locations. The latency of this operation is determined by the slowest placement of a fragment. Therefore, by increasing the redundancy, more distant resources are included in the solution which increases the latency. As a result, the algorithm chooses $m = 3$ redundancy fragments when $A_{req} = 75\%, 79\%, 83\%, 87\%, 91\%$ and $m = 4$ when $A_{req} = 95\%, 99\%$. On the other hand, when the retrieve operation takes place, only k out of $k + m$ fragments are retrieved each time. As a result, increasing the redundancy doesn't burden the latency, but on the contrary, it provides more options for each retrieval.

Finally, when all criteria are equally considered, the results are similar to the case of minimizing the store operation latency which is due to the selection of edge resources whose availability is relatively lower than the availability of the cloud. We expect this case to be somewhere in-between the aforementioned cases, depending on the weight coefficients of the objectives and the characteristics of the infrastructure.

In Figure 28 we examine the effect of the minimum availability requirement on the type of storage resources that are utilized to host the file fragments (cloud or storage resources). All files are split into $k = 5$ fragments and $m = 4$ additional fragments are used for redundancy (erasure code $(9, 4)$). We consider the average utilization among all files. The average number of cloud resources is represented in our graph by the white-colored bars, while the edge resources are represented by the gray-colored bars.

In all samples, when the optimized objective is either the availability, the store and the retrieve monetary costs, or all objectives at the same time, the heuristic utilizes only cloud locations. When the optimized objective is the store and retrieve operation latency, the heuristic utilizes almost only edge resources. This is the case when the minimum required availability is at least 91%. However, in higher required availability (95% sampled), the number of cloud resources progressively increases against the edge resources, although in our samples the number of edge resources is still higher.

This behaviour is explained by the fact that on average, the cloud locations provide higher availability and lower monetary costs than the edge resources. On the other hand, the edge resources are much closer to the end user which provides lower latencies. When the objective is to minimize the store or retrieve latency, in high availability requirements (about 95% or higher), utilizing only edge resources is not enough to meet the user requirement. Therefore, the number of cloud resources increases.

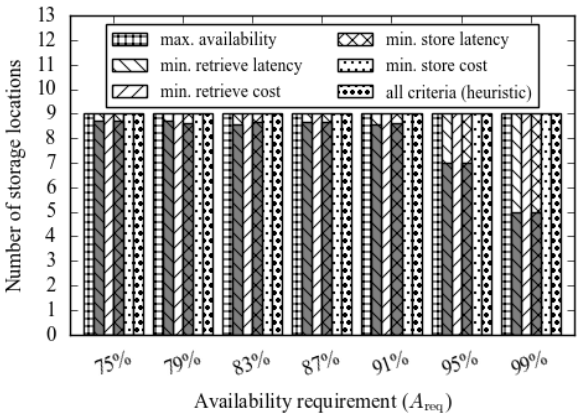

Figure 28: Effect of the minimum availability requirement to the types of the selected storage nodes (heuristic policy).

3.2.4.8 Effect of the optimization objective to the rate of successful retrievals on real data

In Figure 29 we examine the rate of successful file retrievals. A file retrieval is successful when all k storage resources which are selected by the algorithm for each particular retrieval are available when their file fragments are requested for recovery.

The results exhibit that when the criterion is the optimization of the availability, the store and the retrieve monetary cost, the files are always retrieved successfully. This is due to the fact that under the aforementioned criteria, the algorithm chooses only cloud resources. On the other hand, when the optimization criterion is either the store or the retrieve operation latency, the successful file retrieval rate is around 82%, which is interpreted by the fact that edge resources have been selected to host some fragments, which are less available than the cloud resources. Finally, in the multi-objective case, the successful retrieval rate is 94.8% due to the mixture of the selections of both edge and cloud resources.

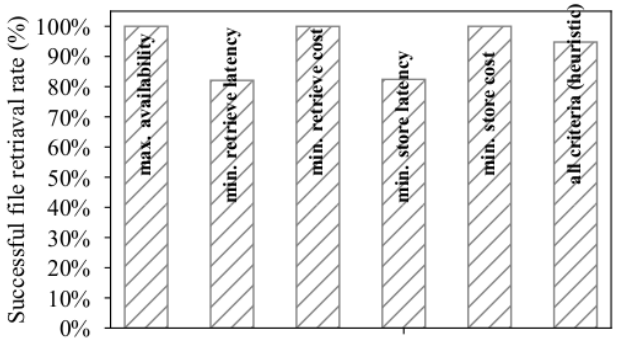

Figure 29: Percentage of successful retrieval for each optimization objective.

3.2.4.9 Progress of the objective values over time on real data

In Figure 30-Figure 34 we examine the progress of the heuristic in terms of cost, to the optimal solution, over the time of the execution. In each graph we consider optimizing with following criteria: a) store operation monetary cost, b) store operation latency, c) retrieve operation monetary cost, d) retrieve operation latency, e) availability and f) all aforementioned criteria at the same time.

In particular, we examine the progress of cost in terms of a) store operation monetary cost (Figure 30, store operation latency (Figure 31), retrieve operation monetary cost (Figure 32), retrieve operation latency (Figure 33) and availability (Figure 34) while optimizing with all the aforementioned criteria separately or simultaneously. The files are split into $k = 5$ and we provide the following options for redundancy to the heuristic: $m \in \{0, 1, 2, 3, 4, 5, 6\}$.

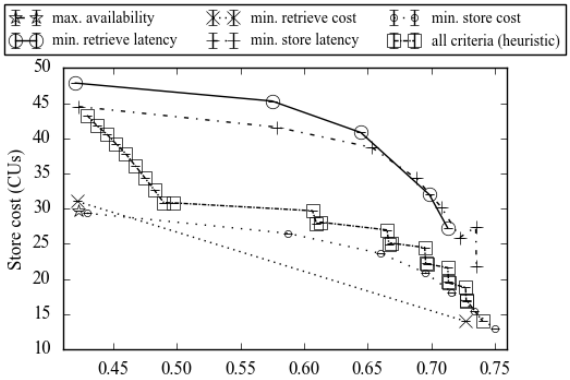

Figure 30: Progress of store operation monetary cost over time.

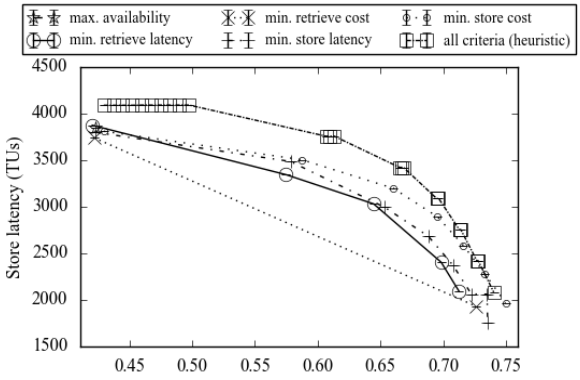

Figure 31: Progress of store operation latency over time.

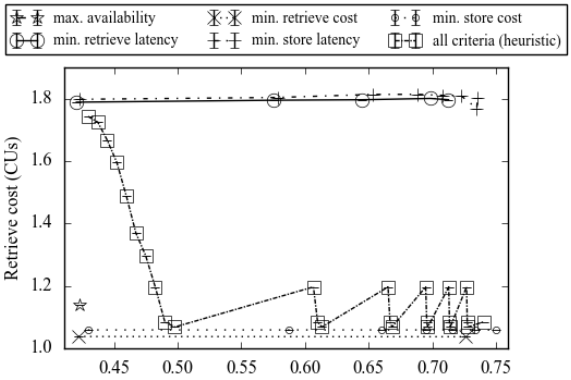

Figure 32: Progress of retrieve operation monetary cost over time.

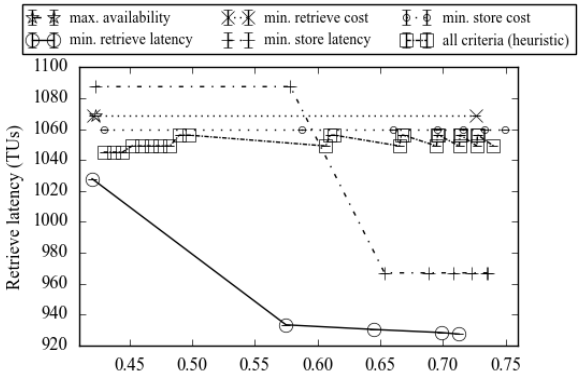

Figure 33: Progress of retrieve operation latency over time.

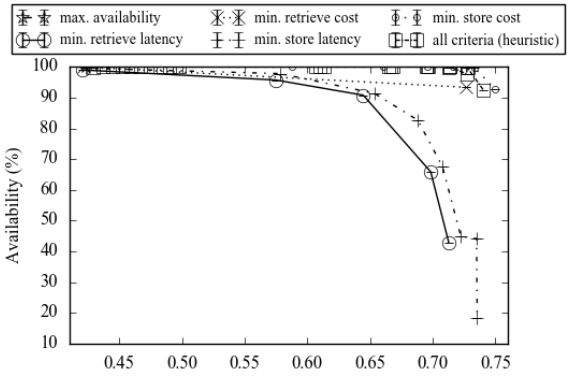

Figure 34: Progress of availability over time.

When the objective is to minimize the store operation monetary cost, the heuristic finds an initial solution at 0.42, which is further optimized 6 times at the following time snapshots: 0.62, 0.66, 0.69, 0.71, 0.73 and 0.75. The corresponding store operation monetary cost at these time snapshots scales from 30 CUs to 27, 24, 21, 18, 15 and 13 CUs. Therefore, achieves an improvement of about 57% in 0.33. At the same, with the same objective the store operation latency does also scale with a similar slope from 3800 TUs to 2000 TUs, which is about 47%. At a first glance, this might seem contradictory. However, the reduction in store monetary cost which also affects the latency is mainly due to the fact that the algorithm gradually chooses solutions with lower redundancy. By reducing the redundancy, the monetary cost is also reduced due to the decrease of the total data size. The latency is

determined from the slowest placement of a fragment, therefore with fewer fragments the store latency tends to reduce. The retrieve operation monetary cost and latency are kept unchanged through the simulations since the redundancy does not affect either the number of fragments or the size of data which is retrieved each time. The availability remains over 95% through time, since the minimization of store cost imposes the utilization of cloud resources.
When the objective is the minimization of the store operation latency, the results are quite similar to the previous case. The heuristic finds an initial solution at 0.42 which is further optimized 7 times at the following time snapshots: 0.58, 0.65, 0.68, 0.71, 0.72, 0.749 and 0.75. The corresponding store operation latency at these time snapshots scales from 3800 TUs to 3500, 3000, 2700, 2500, 2000, 2000 and 1800 TUs. As in the previous case, this is due the fact that the heuristic gradually makes choices of less redundancy, reducing the number of additional fragments. As the number of fragments becomes lower, the maximum distance becomes more limited, which reduces the store operation latency. The retrieve operation cost reduces as well, like in the previous case. The retrieve operation monetary cost is almost unchanged, at around 1.8 CUs per operation, since the number of fragments which are retrieved each time is not directly affected by the redundancy. Finally, for the retrieve operation latency we can observe a drop of about 11% from 1090 CUs at 0.58 to 970 CUs at 0.65. This is due to the fact that initially the heuristic chooses to retrieve the fragments from some distant storage locations since it does not take into account this objective. Then it further reduces the redundancy limiting the maximum distance of a fragment.

Next in Figure 32 we evaluate the effect of minimizing the retrieve operation monetary cost. The initial solution is derived at 0.42 with 1.02 CUs per operation which is slightly improved at 0.73 with 1.02 CUs. As in the previous cases, the heuristic gradually reduces the redundancy. However, the solution is already almost minimum from the beginning, while the reduction of the redundancy does not directly affect the data size and the number of required fragments which are retrieved at each operation. Thus, the cost is only determined by the charges of the chosen storage providers, which is a quite easy task for the heuristic. Furthermore, we observe that the store operation cost is in fact affected by the reduction of the redundancy from 32 CUs to 14 CUs, since the data size which is transmitted during the store operation is reduced, affecting the cost. The store operation latency is also reduced from 3700 CUs to 2000 CUs. The retrieve operation latency seems to be nearly unchanged since it's not directly affected by the redundancy. Finally, the availability remains above 95% which is an indicator that only cloud resources are chosen.

Next, we evaluate the minimization of the retrieve operation latency. Initially, the algorithm begins with a solution with latency 1030 TUs at 0.41. Then the solution gradually improves to 930 TUs at 0.575, 928 TUs at 0.64, 927 TUs at 0.70 and 926 TUs at 0.72, an overall improvement of 10%. This behaviour is due to the fact that initially the heuristic chooses a cloud provider to host a fragment. Gradually, the algorithm reduces the redundancy and leaves only edge resources. The utilization of edge resources reduces the store operation latency and the availability as well, while it keeps the retrieve operation cost nearly unchanged, at around 1.8 CUs per operation. On the other hand, since the total data size

and the total number of fragments counterbalances the relatively high charges of the edge resources, the reduction of redundancy reduces the store operation monetary cost as well.

For the maximization of the availability, the behaviour is quite clear. The heuristic chooses the cloud resources with the highest availability, while utilizing the maximum number of fragments for redundancy ($m = 6$). The solution is found at 0.41 with nearly 100% of availability, and no further improvements are made. Given that $k + m = 11$ cloud locations are selected, the store operation monetary cost of 30 CUs is sensible, while the store operation latency is quite high at 3800 TUs but totally expected. The retrieve operation monetary cost at 1.15 CUs is low and the latency is 1070 TUs which is high. Both of these facts are due to the selection of cloud resources. However, the monetary cost is not minimum and the latency is not maximum since the criterion which is optimized is the availability which does not depend on either the monetary costs or the latency.

Finally, we examine the case where all objectives are optimized simultaneously. In this case, we observe that the store operation monetary cost ranges between the cases of minimizing the latencies and minimizing the monetary costs. The initial solution at 0.42 corresponds to an overall charge of 44 CUs, indicating that the heuristic initially chooses edge resources to host the fragments, with a high level of redundancy. However, it is clear that many cloud locations are also part of the solution since the retrieve operation delay is the highest among all other cases. The heuristic then (between 0.42 and 0.50) sharply reduces the store cost by transitioning to prefer cloud instead of edge resources. This can also be derived from the sharp reduction of retrieve operation monetary cost and the increase of the retrieve operation latency. Both of these metrics nearly reach levels that correspond to the case where the objective is the optimization of store operation monetary costs, 1.05 CUs and 1058 TUs, respectively. In what follows, we can observe that between 0.62 and 0.74, both the store operation monetary cost and the store operation latency gradually decrease. On the other hand, the retrieve operation monetary cost and the retrieve operation latency exhibit fluctuations with a pattern that indicates a tradeoff between them. When an increase is observed in the retrieve operation monetary cost, a decrease is observed in the retrieve latency and vice versa. This behaviour of the heuristic is explained by the fact that the heuristic chooses to retrieve a part of fragments from some edge resources.

3.2.4.10 Effect of the collocation of the storage resources to the optimization objectives on real data

Finally, we examined the effect of node collocation. When the optimization criterion is the store operation monetary cost, the retrieve operation monetary cost and the availability, the results are unaffected by the level of collocation of the edge resources. This is due to the fact that only the cloud resources are chosen for the solution. The cloud resources have lower monetary costs than the edge ones while they provide much higher availability.

When the objective is to minimize the store operation monetary cost, the achieved value is lower than all the other cases, equal to 15.8 CUs. The retrieve operation cost is minimal as well, reaching a cost of 7.4 CUs. On the contrary, the store operation latency is 35932 TUs,

which is the second-highest delay of the results and the retrieve operation latency is 18376 TUs which is the maximal value. Finally, availability is considered, is the second highest, reaching 97.6%. As it was explained before, the heuristic chooses only cloud providers to host the fragments.

Next, we examine the minimization of the retrieve operation monetary cost. The achieved value is nearly equal to the cost of the previous case, reaching 7.4 CUs in all samples for each retrieval. The store operation cost is slightly higher than the previous case, reaching 16.5 CUs, which is an expected result, due to the selection of cloud resources.

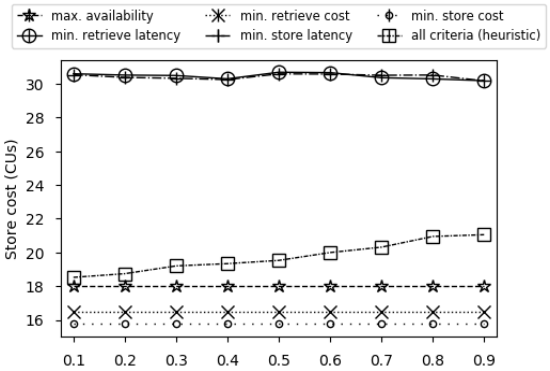

Figure 35: Effect of collocation on store operation monetary cost.

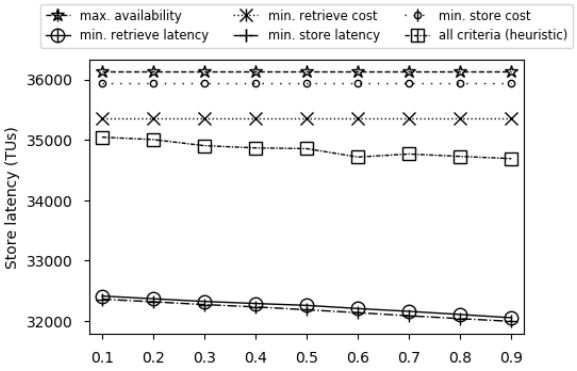

Figure 36: Effect of collocation on store operation latency.

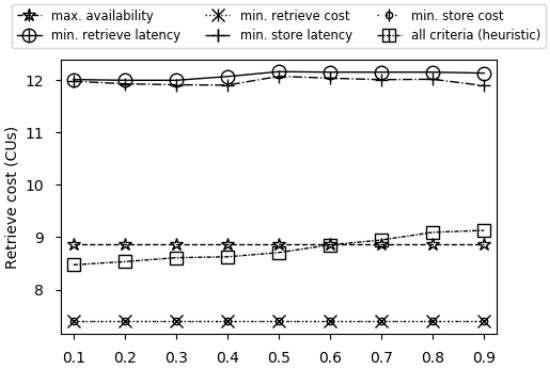

Figure 37: Effect of collocation on retrieve operation monetary cost.

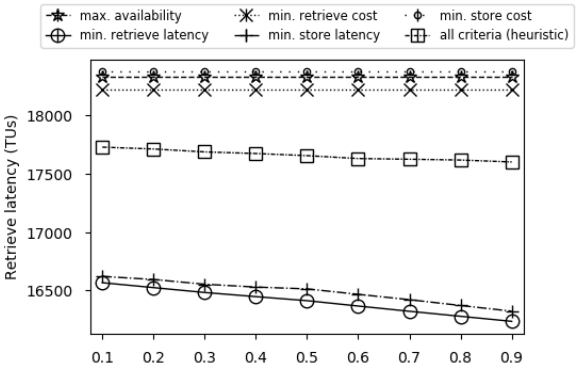

Figure 38: Effect of collocation on retrieve operation latency.

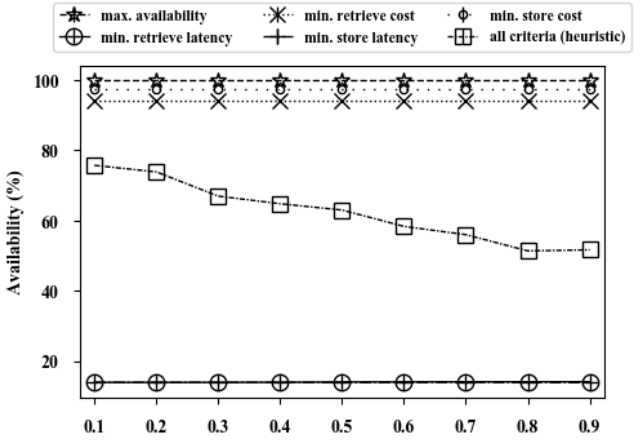

Figure 39: Effect of collocation on availability.

Next, we examined the minimization of the store operation latency. We can observe that the colocation affects the latency. Specifically, as the collocation and thus the edge resources' proximity to the on-premise components increases, the latency decreases. Initially, with a collocation value of 0.1, the latency is 32410 TUs. After that point, the latency linearly decreases until the collocation becomes 0.9 reaching a level of latency of around 32000 TUs. Clearly, that level of improvement is not much. The collocation parameter does only affect the propagation latency but there are other factors that determine the total delay. The first factor is the delay for encoding and encryption that takes place at the on-premise components. This factor depends only on the total size of the transmitted data considering both the initial file sizes and the redundant data. However, in this evaluation scenario, no minimum availability was required and thus the on-premise components were not required to generate additional fragments for redundancy in order to leverage the relatively low availability of the edge resources. We can conclude that if this were the determining factor, at collocation of 0.1, the latency would be around the levels of the previous case where only cloud resources were utilized. This leads us to the next factor, namely the delay of the fragment placement. Both for cloud and edge resources, this factor is determined stochastically. However, we consider the edge to be on average 3 times faster than the cloud resources in terms of data placement delay and thus this factor is the determinant one for the delay reduction. The behaviour is similar when we observe the retrieve operation latency. It starts at 16566 TUs at 0.1 of collocation and reduces linearly to around 16200 TUs at 0.9 of collocation. The store and retrieve operation monetary costs are nearly constant at 30.6 CUs and 12 CUs respectively. Since the monetary costs don't depend on the proximity of the edge resources to the on-premise components, it is expected to be unaffected by the collocation. Any minor fluctuations that are observed in the results, especially in the range of 0.4-0.6 of collocation, are derived by the monetary charging differences between the edge resources which are selected by the algorithm. Finally, the availability is constant at around 14% which is the lowest of all the results, due the utilization of edge resources. The retrieve operation latency exhibits a similar behaviour to the previous case with similar explanations.

The next objective is the maximization of availability. As in previous cases, all metrics are unaffected by the level of collocation due to the utilization of only cloud resources, which are much more available than the edge resources. As was expected, the heuristic achieves the maximal availability 99.9%. As a result, the store operation monetary cost is relatively low at 18 CUs, but higher than the cases where we minimize the monetary costs. Thus, the retrieve operation monetary cost is relatively low at 8.9 CUs per operation.

Finally, when we optimize all criteria simultaneously, we observe the following behaviour. First, the store operation monetary cost increases linearly from 18.5 CUs (collocation of 0.1) to 21 CUs (collocation of 0.9). Accordingly, the retrieve operation monetary cost increases linearly as well from 8.5 CUs (collocation of 0.1) to 9.2 CUs (collocation of 0.9). This behaviour is because heuristic gradually reduces the number of cloud resources in favour of edge resources, as the latter ones' proximity to the on-premise components becomes larger due to the increase of the collocation. This increase of the utilization of edge resources is clear by inspecting the results of the availability metric, which reduces linearly from 75.7%

(collocation of 0.1) to 52% (collocation of 0.9). The increase of the proximity benefits the latency objectives of the multi-objective optimization allowing the utilization of the relatively more expensive edge resources. Finally, the store and the retrieve operation latencies are observed to be affected by the increase of collocation. The store operation latency reduces from 35042 TUs to around 34700 TUs at collocations of 0.1 and 0.9, respectively. Accordingly, the retrieve operation latency reduces linearly from 17729 TUs to around 17100 TUs, at collocations of 0.1 and 0.9, respectively. The improvement is achieved by the decrease of cloud in favour of edge resources, which are getting more approximate to the on-premise components.

3.2.5 Conclusions

The amount of data created, consumed, and stored is expected to increase in the following years dramatically. This fact makes the distributed storage infrastructures and services critical components for the targeted digital transformation of our society. Edge computing is expected to improve the performance of these systems, while erasure coding techniques increase the security and the availability of data. In this work, we proposed resource allocation mechanisms that take into account these aspects when allocating storage resources for the data fragments that result from the erasure coding operation. As is shown by the simulation experiments, based on actual data, the developed mechanisms efficiently explore the different characteristics of the resources in an edge-cloud infrastructure, trading-off performance with execution time. Furthermore, the performed simulation results illustrate the advantages in monetary cost and latency that these mechanisms offer when multiple criteria are simultaneously taken into account during the optimization.

4 Resource Optimization Toolkit

The SERRANO platform has to automatically determine the most appropriate resources across the distributed edge/cloud/HPC infrastructure to be used for the deployment of the applications. The Resource Optimization Toolkit (ROT) integrates the designed resource allocation algorithms (Section 3) in the SERRANO platform, implementing the decide part at the envisioned closed-loop control based on the principles of observe, decide and act.

The ROT provides to the SERRANO Resource Orchestrator (detailed description available at deliverable D5.3 “Resource Orchestration, Telemetry and Lightweight Virtualization Mechanisms”) the required logic to allocate the edge, cloud and HPC resources so as to satisfy the applications’ requirements, to coordinate the efficient movement of required data across the selected resources and to support proactively and reactively re-optimization adjustments.

4.1 ROT functional architecture

The Resource Optimization Toolkit implements the developed multi-objective optimization and orchestration algorithms. Moreover, it prepares, coordinates, and manages the appropriate algorithm's execution to facilitate the Resource Orchestrator to its application deployment and service assurance functionalities. ROT has been designed to provide fast execution and efficient resource usage while scaling with the demands. Furthermore, a primary concern during its design was to build a system that could be easily extendable with new algorithms.

Figure 40 shows the architecture of the ROT, its main components, and the interactions with other components within the SERRANO architecture. According to the selected design, there is one Dispatcher but multiple workers. Each worker is composed of the Execution Engine and the library of the decision algorithms. This approach ensures that the ROT will always be able to handle quickly any number of execution requests by the Resource Orchestrator, even in very complex infrastructures.

The Access Interface provides loose coupling between the Resource Optimization Toolkit and the other components within the SERRANO platform. It exposes the necessary interfaces that allow the bidirectional communication for exchanging commands, information and notifications. More details for the provided methods are available in Section 4.2.

The Dispatcher provides the needed logic and information model adaptations to manage the execution of the requests and to handle the interaction with the multiple instances of the Execution Engine. It is invoked by the Access Interface whenever there is a request for executing resource allocation computations through the SERRANO orchestration algorithms. It also interacts with the Central Telemetry Handler and the operational and monitoring databases to retrieve the characteristics of the resources, their current status and the deployed applications. The Dispatcher prepares the required input parameters and

distributes the execution requests on the available Execution Engines to obtain the results of the executed algorithms in a reasonable amount of time.

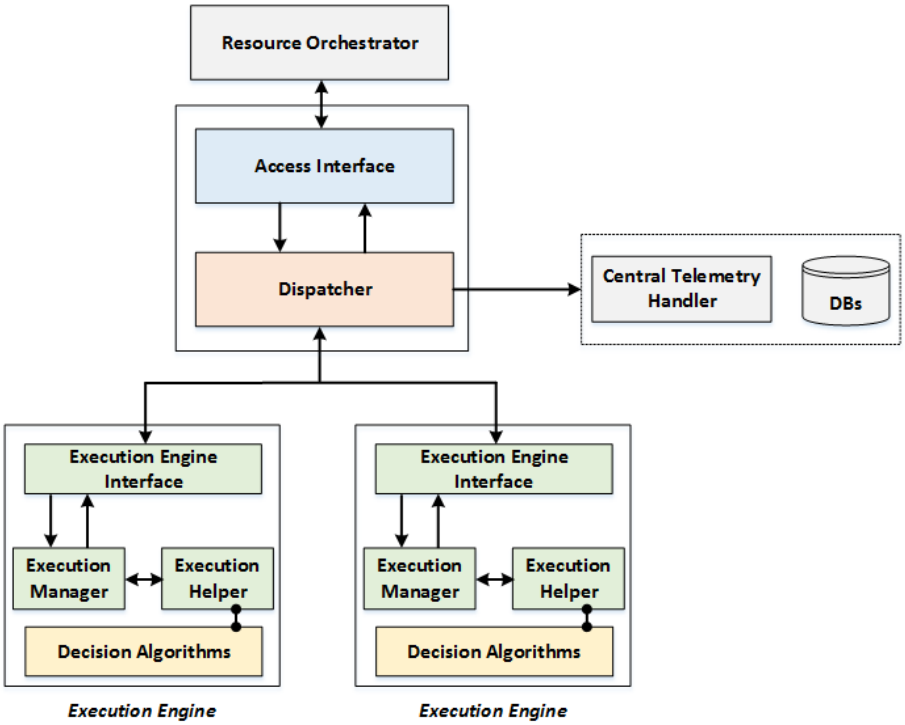

Figure 40: Resource Optimization Toolkit architecture and main components

The Execution Engine receives instructions, for starting or terminating algorithm executions, from the Dispatcher and performs all the required actions, including the preparation of the execution environment, the monitoring of the execution progress and the handling of the final results or possible failures. Furthermore, it is responsible to monitor the resources of the node where it is executed and return related information. Each service request is treated as a separate task by the Execution Engine and corresponds to either terminating a particular execution or starting a new one. In the latter case, the service request includes also all the necessary input parameters.

The Execution Engine contains the following components:

- ***Execution Engine Interface:*** receives and validates service requests from the Dispatcher and forwards them to the appropriate internal component.
- ***Execution Manager***: handles the execution of the service requests, either by terminating a running task or by starting a new execution with the help of Execution Helper. It collects also the required monitoring information and informs the Dispatcher accordingly.
- ***Execution Helper:*** prepares the execution of the requests by setting up the execution environment, monitors the progress of the tasks and handles the output results or possible failures.

- **Decision Algorithms**: the library of multi-objective optimization and orchestration algorithms.

The main components of the ROT software are implemented in Python, using additional frameworks like Flask 2.0 [22], Pika [23] and PyQt [24]. In addition, it internally uses an SQLite database. The resource allocation algorithms will be implemented either in C++ or in Python. They will be accessible from the Execution Engine internal components through a custom plug-in mechanism. This mechanism will expose a common interface, independent of the implementation technology and algorithm internal logic, that among others will determine the explicit syntax of the input parameters and the results for all algorithms. In addition, the common interface will use the JSON as data-interchange format for providing the input and output parameters.

Next, we provide a detailed description (Figure 41) of the workflow for executing resource allocation algorithms within the SERRANO platform, highlighting the roles of the ROT components and their interaction with other services of the SERRANO architecture.

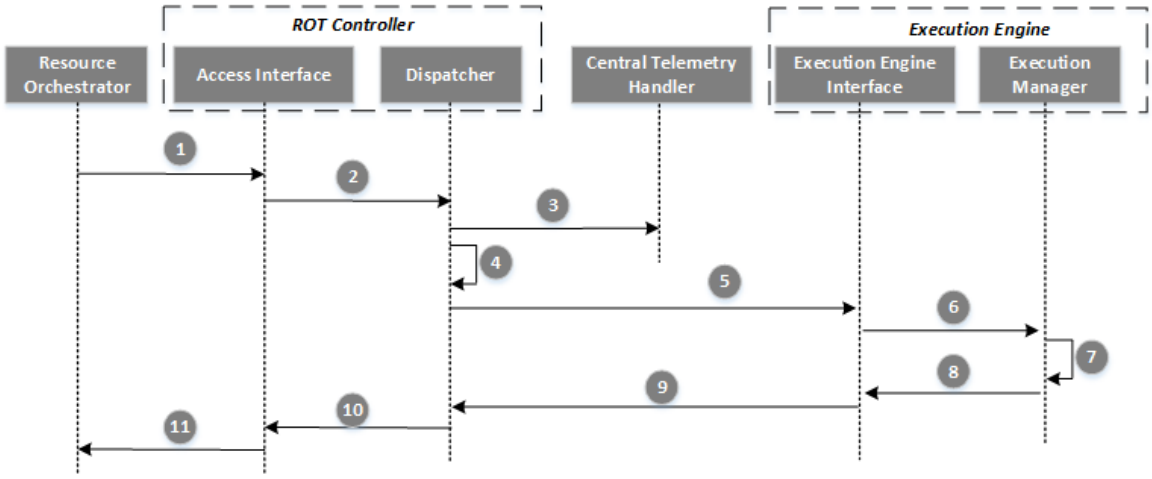

Figure 41: ROT – Workflow for resource allocation computation

When the Resource Orchestrator needs to decide for the initial deployment of an application or to reconfigure an existing one (after being triggered by the service assurance mechanisms), an execution request is sent to the ROT through the Access Interface (Step 1). Next, the Dispatcher prepares the input parameters and maps the high-level requirements to the appropriate SERRANO resource profiles (Step 2). To this end, it interacts with the telemetry components and the databases that provide the operational and monitoring data (Step 3). Then, it assigns the execution request to the Execution Engine with the least workload (Step 4). The selected Execution Engine retrieves the request (Step 5), and its internal components setup the execution environment (Step 6) and run the selected algorithm with the provided input data (Step 7). The Execution Helper and Execution Manager monitor the progress of the execution requests (Steps 7-8) and send the resource allocation decisions to the Dispatcher (Step 9). Finally, the ROT forwards the output to the

Resource Orchestrator that continues with the required operations for the actual deployment of the applications (Steps 10-11).

4.2 Interfaces

The ROT offers two main North Bound Interfaces (NBIs), the first is based on REST APIs over the HTTP protocol and the second is an asynchronous messaging interface based on the Advanced Message Queuing Protocol (AMQP). The former exposes control operations to manipulate and inspect the execution of deployment algorithms. The latter offers persistence and asynchronous communication between ROT and the Resource Orchestrator to send responses and notification messages.

Table 7 summarizes all the supported methods for the REST-based interface, while the following tables present for each method the list of the information elements included in the body of the HTTP request or response and the possible HTTP response codes. The REST API methods are implemented in Python based on the Flask 2.0 micro web framework, with the async module enabled to support many concurrent IO-bound requests efficiently. In addition, all the REST APIs include Swagger documentation (Figure 42).

Table 7: Resource Optimization Toolkit REST API

HTTP Verb	URI	Description
GET	/api/v1/rot/executions	Get the list of all active executions.
POST	/api/v1/rot/execution	Start the execution of some specific algorithm with the requested input parameters.
GET	/api/v1/rot/execution/{uuid}	Get the details of a specific algorithm execution
DELETE	/api/v1/rot/execution/{uid}	Terminate a specific algorithm execution.
GET	/api/v1/rot/statistics	Get statistics for the completed executions.
GET	/api/v1/rot/engines	Get the available execution engines.
GET	/api/v1/rot/engine{uuid}	Get details about a specific execution engine.
GET	/api/v1/rot/logs/{uuid}	Get detailed logging information for a specific algorithm execution.

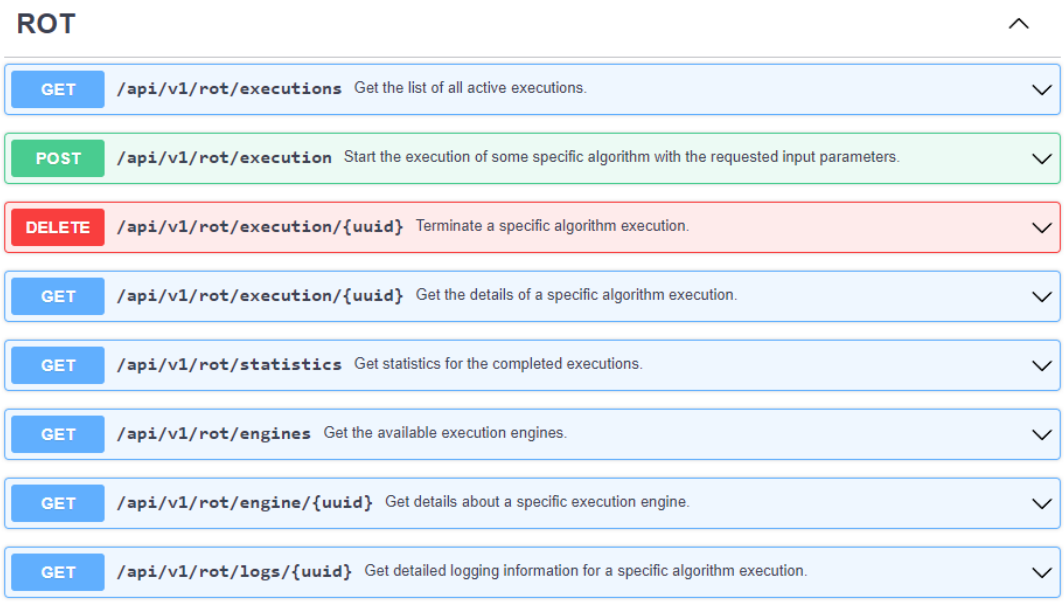

Figure 42: ROT access interface – Swagger documentation of REST APIs

Table 8: GET /api/v1/rot/executions

GET /api/v1/rot/executions			
Request Parameters	--	--	--
Response Parameters	executions	List of elements	Each element provides details for an active execution. The parameters are listed in Table 9.
Successful response HTTP code: 200 OK			
Error response HTTP code: 400 BAD REQUEST, 403 FORBIDDEN, 409 CONFLICT 500 INTERNAL SERVER ERROR, 503 SERVICE UNAVAILABLE			

Table 9: GET /api/v1/rot/execution/uuid

GET /api/v1/rot/execution/uuid			
Request Parameters	--	--	--
Response Parameters	execution_id	String	The unique identifier of the execution request.
	engine_id	String	The unique identifier of the execution engine.
	status	Enum	The current status of execution request (same values with event EXECUTION_INFO at Table 16).
	results	Element	If the execution is completed successfully (status: DONE) provides the results, otherwise is empty.
	created_at	Integer	Timestamp indicating when ROT received the specific execution request.
	updated_at	Integer	Timestamp in when receives the specific execution request.
Successful response HTTP code: 200 OK			
Error response HTTP code: 400 BAD REQUEST, 403 FORBIDDEN, 409 CONFLICT 500 INTERNAL SERVER ERROR, 503 SERVICE UNAVAILABLE			

Table 10: POST /api/v1/rot/execution

POST /api/v1/rot/execution			
Request Parameters	execution _params	Element	Includes a list of parameters that describe the specific input parameters for the selected algorithm.
	algorithm_module	String	Name of the module that provides implementation of the algorithm.
Response Parameters	execution_id	String	The unique identifier of the execution request.
	status	Enum	The status of the request: • Accepted • Rejected • Failed
Successful response HTTP code: 201 CREATED			
Error response HTTP code: 400 BAD REQUEST, 403 FORBIDDEN, 409 CONFLICT 500 INTERNAL SERVER ERROR, 503 SERVICE UNAVAILABLE			

Table 11: DELETE /api/v1/rot/execution/uuid

DELETE /api/v1/rot/execution/uuid			
Request Parameters	--	--	--
Response Parameters	--	--	--
Successful response HTTP code: 200 OK			
Error response HTTP code: 400 BAD REQUEST, 403 FORBIDDEN, 409 CONFLICT 500 INTERNAL SERVER ERROR, 503 SERVICE UNAVAILABLE			

Table 12: GET /api/v1/rot/statistics

GET /api/v1/rot/statistics			
Request Parameters	start (Optional)	Integer	Timestamp specifying the starting point of the data to be returned.
	end (Optional)	Integer	Timestamp specifying the ending point of the data to be returned.
Response Parameters	execution_ requests	Integer	Total number of execution requests.
	completed_executions	Integer	Successfully completed executions.
	failed_executions	Integer	Number of failed executions.
	avg_exec_time	Integer	Average execution time in secs.
	number_of_engines	Integer	Number of used execution engines.
	avg_exec_time_per_engine	Element	Includes a list of average execution time in each utilized execution engine.
Successful response HTTP code: 200 OK			
Error response HTTP code: 400 BAD REQUEST, 403 FORBIDDEN, 409 CONFLICT 500 INTERNAL SERVER ERROR, 503 SERVICE UNAVAILABLE			

Table 13: GET /api/v1/rot/engines

GET /api/v1/rot/engines			
Request Parameters	--	--	--
Response Parameters	engines	List of stings	Correspond to unique identifiers of the available execution engines.
Successful response HTTP code: 200 OK			
Error response HTTP code: 400 BAD REQUEST, 403 FORBIDDEN, 409 CONFLICT 500 INTERNAL SERVER ERROR, 503 SERVICE UNAVAILABLE			

Table 14: GET /api/v1/rot/engine/uuid

GET /api/v1/rot/engine/uuid			
Request Parameters	--	--	--
Response Parameters	engine_id	String	Engine unique identifier.
	hostname	String	Hostname.
	uptime	Integer	Total number of seconds that execution engine is running.
	active_executions	Integer	Number of active executions.
	total_executions	Integer	Total number of executions.
	failed_executions	Integer	Total number of failed executions.
Successful response HTTP code: 200 OK			
Error response HTTP code: 400 BAD REQUEST, 403 FORBIDDEN, 409 CONFLICT 500 INTERNAL SERVER ERROR, 503 SERVICE UNAVAILABLE			

Table 15: GET /api/v1/rot/logs/uuid

GET /api/v1/rot/logs/uuid			
Request Parameters	--	--	--
Response Parameters	log_details	Element	Includes a list of trace information, organized based on the different phases of the execution lifecycle.
Successful response HTTP code: 200 OK			
Error response HTTP code: 400 BAD REQUEST, 403 FORBIDDEN, 409 CONFLICT 500 INTERNAL SERVER ERROR, 503 SERVICE UNAVAILABLE			

The SERRANO platform relies on a message broker-based interface to collect and forward asynchronously the appropriate messages and events from the various distributed components. This interface manages notifications messages between distributed components and software modules in a scalable and efficient way. To this end, SERRANO incorporates in the overall platform the Data Broker component to support both message brokering and distributed streaming capabilities through the abstraction and integration of the appropriate message infrastructure. The respective technical presentation is provided in

deliverable D5.3 “Resource orchestration, telemetry and lightweight virtualization mechanisms” (M15).

In particular, the Data Broker provides a predefined message queue through which the SERRANO Resource Orchestrator and the Resource Optimization Toolkit can asynchronously exchange messages. The ROT posts messages to the predefined exchange that the Resource Orchestrator uses to coordinate the necessary actions to trigger the necessary actions for the actual deployment based on the decisions of the ROT. These messages are described in JavaScript Object Notation (JSON) format using the following predefined syntax:

- ***event*** (*string*): event unique identifier.
- ***data*** (*element*): set of event related parameters that provide the required information.

The following table describes the structure of the notification messages exposed by the ROT.

Table 16: Resource Optimization Toolkit notification messages

Event Identifier	Description	Parameters
ENGINE_INFO	Dispatcher scaled up the number of available execution engines.	• ***engine_id*** (*string*): execution engine unique identifier • ***message*** (*string*): event related information • ***timestamp (****integer)*: Unix time stamp
ENGINE_FAILED	Specific execution engine encountered operational issues.	• ***engine_id*** (*integer*): execution engine unique identifier • ***message*** (*string*): event related information • ***timestamp (****integer)*: Unix time stamp
ENGINE_DOWN	Dispatcher scaled down the number of available execution engines.	• ***engine_id*** (*string*): execution engine unique identifier • ***message*** (*string*): event related information • ***timestamp (****integer)*: Unix time stamp
EXECUTION_INFO	Provide information regarding the current state of a specific execution request.	• ***uuid*** (*string*): execution request unique identifier • ***message*** (*string*): event related information • ***status*** (*string*): current status of execution request: ○ PENDING: execution request is prepared, pending assignment at specific execution engine ○ STARTED: execution request is running ○ FAILED: execution request failed ○ DONE: execution request finished

		successfully ○ TERMINATED: execution request terminated • ***timestamp (****integer)*: Unix time stamp
EXECUTION_RESULTS	Execution request is successfully executed and the ROT provides the results.	• ***uuid (integer):*** execution request unique identifier • ***results*** *(string):* algorithm execution output • ***timestamp*** *(integer):* Unix time stamp

5 Energy and Resource Aware Flow Mapping

In the current phase of Task 5.4, we are focusing on preparing the testbed, the Excess cluster, which is intended both to measure the energy efficiency of HPC services and to serve as a test platform for the SERRANO orchestrator. The testbed provides us more opportunities to find the best configuration of HPC services from the point of view of performance and energy efficiency and for the preparation of the Hawk supercomputer usage. This preparation includes hardware installation, tools development and execution, and analysis of the benchmarks in order to investigate the behaviour of the hardware and software that is used in the SERRANO project, in particular for HPC Services.

5.1 Testbed Excess Cluster

Figure 43 shows the main components of the EXCESS cluster:

- Login node is a Gateway to the compute nodes of the cluster.
- Node01 is a compute node that has the same CPU model as the Hawk supercomputer. Six sensors are installed in the node to measure the electric power and voltage at three 12-volt power sources. This allows the power consumption of the main memory (64 GB), the CPU, the voltage regulators and InfiniBand Adapter to be recorded.
- Addi is a server with AC/DC converters, which are connected to the sensors of node01. The measurement is started for each compute job and stored in the home directory after the job is finished. The recorded data can be used for further energy profiling and analysis.

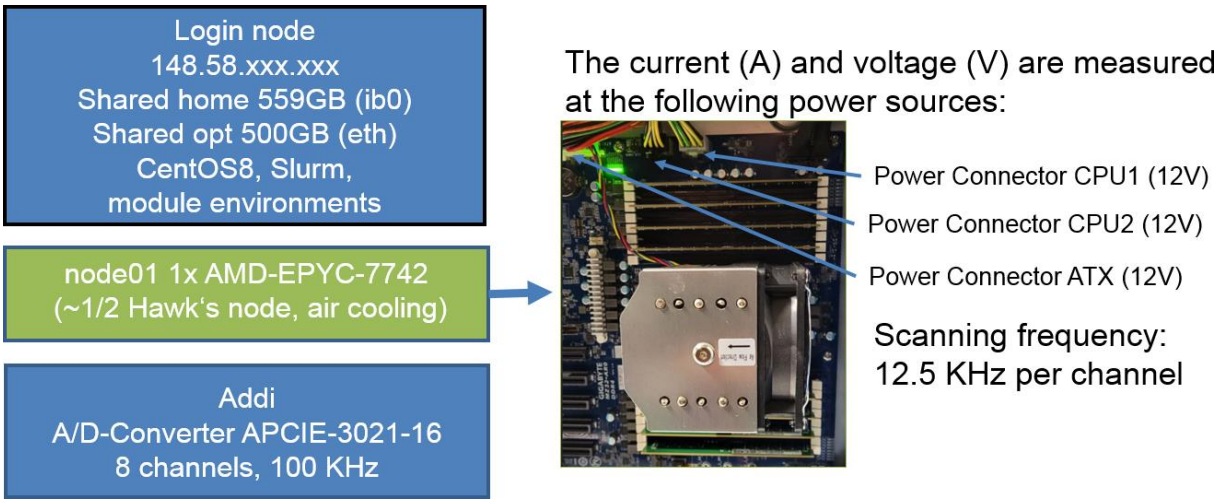

Figure 43: Hardware components of the Excess cluster

The relative accuracy of the external power measurements is less than 0.1 percent. The absolute accuracy of less than one percent is achieved after calibration of its sensors. Further details about the system for measuring power consumption in the EXCESS cluster are available in the PDF documents of the Github project under https://github.com/excess-project/cluster-power-tools/tree/master/power-measurement-system.

5.2 Testbed Excess Cluster – Hardware and Tools

One of the reasons for compute node "node01" installation and its connection to the power measurement system is that the Hawk supercomputer does not have an interface to measure the power consumption. Hence, it is impossible to investigate the various possible configurations of HPC services according to the energy efficiency.

Although the compute nodes of the Excess and Hawk are not identical, as it is shown in Table 17, the energy efficiency of these two systems can essentially be compared. However, one of the prerequisites for this is that the operating mode of the processors and the memory are configured similarly as much as possible. Therefore, the BIOS and OS settings were performed together with the AMD staff supporting the work of the Hawk supercomputer at HLRS according the "AMD HPC Tuning Guide for AMD EPYC™ Processor"[2] [42].

Table 17: Comparison between compute nodes of Excess and Hawk

Hardware	Excess compute node	Hawk compute node
CPUs	1 x AMD-EPYC-7742	1 x AMD-EPYC-7742
Main memory	Samsung SDRAM DDR4 double rank, 3200 MT/s, 8x16GiB	Micron SDRAM DDR4 double rank, 3200 MT/s, 16x16GiB

The Rome processor AMD-EPYC-7742 was introduced in 2019. The CPU has 64 cores, 128 HW threads, 256 MB shared L3 cache and can be connected to the SDRAM-DDR4 modules over eight memory channels. The memory channels can be clocked with 3200 MHz.

Unlike the processor of the previous generations, especially Intel CPUs, the AMD processor EPYC™, so-called "Rome", consists of several chips, as it is schematically represented in Figure 44. Eight chips with up to 8 cores each, also called chiplets, are interconnected via a central I/O chip, so-called "Infinity Fabric". The use of multiple chips in a single package has both disadvantages and advantages:

- The omission of the large chips simplifies the adoption of the latest semiconductor technology. AMD used the 7 nm TSMC semiconductor technology for the production of the chiplets of the Rome processor, also called "Compute Core Dies" (CCD). This is claimed to be comparable with 10 nm Intel semiconductor technology [43].
- The CPU's pads and bond islands are expensive (needs a lot of splace) and cannot be attached to a chip in an arbitrary number. For this reason, it is a challenge to

[2] The first experiments were also performed to tune the production mode of the Hawk supercomputer.

implement a scalable and fast interconnect for the chiplets. AMD has implemented it with the help of an additional I/O chip, so-called an "Infinity Fabric" that interconnects the CCDs with each other and with system components outside the CPU package, such as main memory. The fact that the I/O chip is manufactured with 14 nm technology additionally highlights the great complexity that the process manufacturers have to face in the further miniaturization of semiconductor technology.

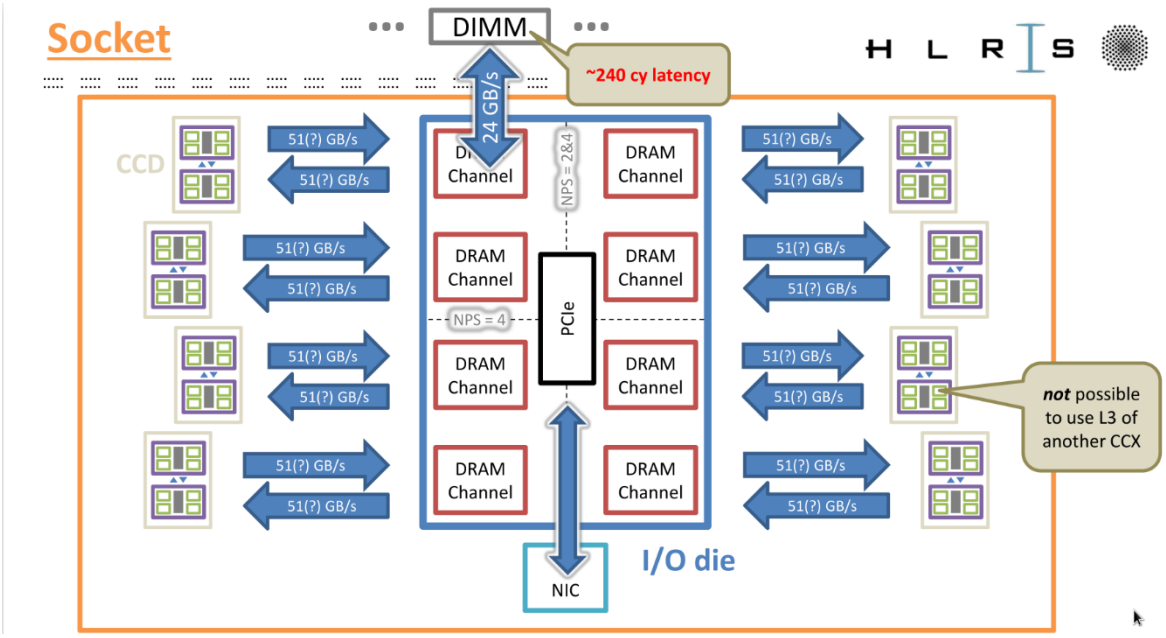

Figure 44: The AMD EPYC Rome processor

The CPU package is inserted into a CPU socket on the mainboard. In this way, the external pins of the CPU package are connected to the various components, such as the main memory interface over eight internal memory controllers (IMC), labeled as DRAM Channel in Figure 3 and the PCIe interfaces of the motherboard. The InfiniBand Adapter, Connect-IB HCA Mellanox MCB193A-FCAT, single-port QSFP is also connected to the processor of the node01 (labeled as NIC in Figure 44).

On the one hand, the external measuring system we use cannot measure the consumption of the single components and on the other hand, the embedded hardware counters “Running Average Power Limit” (RAPL) cannot measure the power consumption of the memory modules and other external components.

The accuracy of RAPL measurements is not specified. We estimate it at 10% percent. Unlike the Hawk supercomputer, we have the possibility on the Excess cluster to read the RAPL counters that need root access.

We also have the possibility to change the CPU frequency on the node01. The processor can be set with four different frequencies. Currently it is done with the help of the “cpupower” tool [44] in the installed bash script. The processor can be operated in four different

frequency modes. Here are the available modes and the corresponding configuration parameters of the “cpupower” tool:

- **1.5 GHz:** BOOST = 0, MIN_FREQ=1.5 GHz, MAN_FREQ=1.5, GOV = userspace;
- **2.0 GHz:** BOOST = 0, MIN_FREQ=2.0 GHz, MAN_FREQ=2.0, GOV = userspace;
- **2.25 GHz:** BOOST = 0, MIN_FREQ=2.25 GHz, MAN_FREQ=2.25, GOV = userspace;
- **Turbo Mode:** BOOST = 1, MIN_FREQ=2.25 GHz, MAN_FREQ=2.25, GOV = userspace;

Since that bash script needs root privileges, additional entries are made in the system file “/etc/sudoers” so that users can start the script under root:

```
Username ALL=(ALL) NOPASSWD:SETENV: /opt/power/rome-freq/bin/set_node01_frequency.sh
```

5.3 Peak Performance and Example Pursuit of Peak

High Performance Computing (HPC) addresses the various types of algorithms. A "general-purpose CPU" is particularly efficient for computing at a low rate $\frac{Byte}{Flop}$. The peak power therefore corresponds to $Byte = 0$, if all operands and results of the floating point operations are stored in the data registers and there are no data dependencies. The state-of-the-art processors keep the floating point numbers in special registers, namely vector registers. In a vector register multiple floating point numbers can be held and processed in parallel: therefore there is the requirement of data independence.

The width of the vector registers in the x86 architecture has not really grown significantly in the last 20 years. Intel's first streaming SIMD extension (SSE), introduced in 1999, supports vector registers with two 64-bit double precision floating point numbers (doubles). Figure 45 shows a schematic representation of one of the AVX micro operations, namely addition with two operands. The FMA extension additionally supports Fused Multiply Add (FMA) micro operations, which perform addition and multiplication in one turn (three operands). Both AVX and FMA micro operations are implemented in different variants. For example, a bit mask can be used to exclude certain elements of the vector registers from the operation.[3]

A Haswell processor "E5-2680v3" from the former supercomputer "Hazel Hen" at HLRS with the peak performance of ~7.4 PFlops supports the AVX2- extension and, as a superscalar processor with two FMA units, can execute up to two FMA micro commands simultaneously. Hence, the Haswell with 12 cores and the base frequency of 2.5 GHz has a peak performance of 12 x 2 x 2 x 4 flops x 2.5 GHz = 480 GFlops.

[3] In a Bash environment, the supported CPU extensions can be monitored with the command with the command cat /proc/cpuinfo.

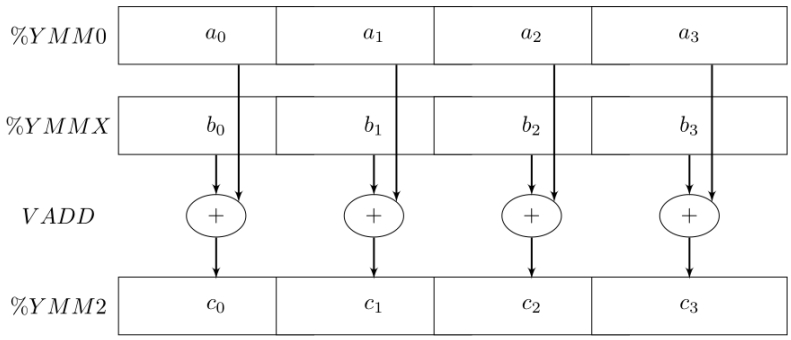

Figure 45: Execution of a single instruction multiple data (SIMD) addition on AVX registers

The next generation Skylake server processors "Intel Xeon Platinum 8168" support the "Intel Advanced Vector Extensions 512"(AVX-512). AVX-512 is intended for the vector registers which can store up to 8 doubles. Thus, superscalar processor of the supercomputer "JUWELS" at the JSC [45] has a peak performance of 28 x 2 x 8 flop x 2.7 GHz = 2.074 TFlops. The theoretical performance of the "JUWELS" is therefore ~12 PFlops.

The latest AMD processor "Rome EPYC 7742" of the next HLRS's Super computer "Hawk" [46] does not support AVX-512, but it has 64 superscalar cores with AVX2 and has a theoretical peak performance of 64 x 2 x 2 x 4 flop x 2.25 GHz = 2,304 TFlops. The theoretical performance of the Hawk is 26 Petaflops.

For comparison: The NEC vector engine processor "SX-Aurora (Type 10A)" has only 8 cores, which are clocked with a relatively low frequency of 1.6 GHz. Nevertheless, the vector processor has a peak performance of 2.45 TFlops. An important feature of vector processors is that their frequency always remains constant, which ensures efficient operation. However, one of the prerequisites for this is that the code is vectorized. This is not always possible, since the vector registers are of length 2048 bits [47][48]. The AMD processor "Rome EPYC 7742, which we are currently focusing on, so that the frequency reaches higher values than the base frequency.

The use of the vector registers and the corresponding part of the arithmetic unit requires more electric power and consequently generates more heat than the scalar operations. For this reason, the processors regulate the operating CPU frequency down during the computation. The CPU frequency may well fall below the base frequency, as it is for example especially the case with AVX-512 expansion.

5.3.1 Benchmark Pursuit of Peak

In this benchmark, an "artificial" algorithm is considered, during the execution of which data is kept in the registers to eliminate delays due to data transport to and from cache and main memory. One of the objectives of this benchmark is to compare the configuration

parameters of the CPUs in the Excess cluster and in the Hawk. Several FMA[4] commands are executed one after the other in the loop. The loop is unrolled by factor 8 and will be executed in parallel with n threads (n is variable):

```
L3:
# for(ii=0; ii<loop_length; ii+=32)
        addq    $32, %rax       #, ii
        vfmadd231pd     %ymm1, %ymm0, %ymm2     # _56, _166, CYMM
…
        vfmadd231pd     %ymm1, %ymm0, %ymm6     # _56, _166, CYMM7
        cmpq    %rax, %rdi      # ii, loop_length
        jg      .L3     #,
```

Table 18 shows the results of the “Pursuit of Peak” benchmark on AMD Rome Processor on one and 64 cores of the Excess and Hawk. The theoretical possible peak performances of the CPUs are specified for one and 64 cores and for 2.5 GHz and 3.4 GHz. The highest CPU frequency specified by the manufacturer. The next two columns show the achieved performance on both systems. As the reader can see, the performance on the Hawk varies more than on Excess. The reason for this is that the benchmark was started several times in the PBS jobs and, in contrast to the Excess Cluster, the jobs are executed on different compute nodes of the Hawk. The small differences, for example in the liquid cooling temperature, account for these differences.

It is worth noting that the highest achieved frequency in both systems is 2.71 GHz. The reason for this is that the CPUs of the Excess and the Hawk have the similar BIOS settings and its power budgets are limited, although the cooling systems are different: The Hawk uses liquid cooling and the Excess cluster uses air cooling (the server room has a constant temperature around 22° C).

Another interesting fact is that the power consumption hardly changes in both the RAPL and the external power measurements, regardless of whether it is calculated with one or all 64 cores.

The larger differences in the external power measurement between one and 64 cores can be related to the fact that the power consumption of the memory modules is also measured. Even if no data is transported between the memory and the cores, a certain activity, e.g. related to memory coherence, occurs in the IMC that increases with more cores.

[4] The FMA extension supports the fused multiply add operation (assembler: vfmadd213pd) on the 256-bit vector registers (assembler: %ymm). The FMA operation performs addition and multiplication in one turn. Furthermore, in ROME CPU several FMAs can be performed at the same time.

Table 18: Results of the benchmark "Pursuit of Peak" on AMD Rome Processors on one and 64 cores of the Excess Cluster and Hawk Supercomputer

#Cores	Th. Peak [TFLOPS]	Excess [TFLOPS]	Hawk [TFLOPS]	CPU Freq.	RAPL [W]	Ext. Power [W]	Diff. [W]
1	0.036 / 0.0544	0.0433-0.0434	0.0430-0.0432	~2.71	76.84	115.26	>30.7 <46.1
64	2.304 / 3.482	2.775-2.777	2.685-2.745	~2.71	76.88	120.07	>35.7 <51.1

The differences between the RAPL and the external power measurement (column "Diff."[5]) also give us an idea of the power consumption of the individual hardware components. The previous experiments with the power measurement show that the power consumption of the used InfiniBand adapter is 4.8 watts[6] [49].

It is important that the electrical power of the remaining components be considered as well. This will determine which of the configurations under consideration, such as pinning strategy, CPU frequency and number of cores, is the most efficient from an energy consumption point of view [50].

Fortunately, this share of the electrical power changes very insignificantly compared to the consumption of the memory modules, processor and the voltage regulators. According to previous observations, we can assume that the rest of the hardware does not consume more than 50 watts. It is intended that the exact total power consumption of the node01 will be determined during the duration of the project with additional equipment [51].

The corresponding power consumption of a Hawk's node has to be discussed with the manufacturer, namely HPE. This also includes the energy consumed for the infrastructure of the supercomputer, such as cooling. This approach was successfully approved in the approximation of the energy consumption of the previous supercomputer at HLRS, namely Hazel Han [52].

5.3.2 Memory Bandwidth and Example Sustained Bandwidth

According to Moore's Law, the complexity of integrated circuits regularly doubles. The consequence of this is a significant increase in the number of cores. Therefore, the processor architecture must increase the internal parallelization level accordingly.

For example, keeping up the shared cache performance is only guaranteed by a sophisticated topology and increasing the frequency of interconnect, which is becoming more complex and from one micro- architecture generation to the next add new constraints. In the AMD Rome processor, for example, the L3 caches are only shared between 4 cores of a CPU Complex (CCX) (see Figure 44).

[5] The ten percent of RAPL accuracy is included.
[6] The power approximation for the case without traffic via InfiniBand.

In the same way, the main memory performance does not increase as fast as the theoretical peak performance of the CPU. Although the data transport between the memory modules and the processor increases, it decreases per core. One of the limiting factors for this is, that the number of memory channels and IMCs does not grow proportionally to the number of cores. The performance increase of the individual memory modules cannot compensate for this shortcoming.

The number of memory channels, its bus width and its frequency are used for the calculation of the theoretical bandwidth.

Table 19: Theoretical bandwidth

Processor	Num. memory ch.	Width [B]	Memory frequency MHz	Theor. Bandwidth GB[7]/s
Haswell E5-2680v3	4	8	2133	68.256
AMD EPYC 7742	8	8	3200	204.8

5.3.3 Example Sustained Memory Bandwidth

In this benchmark, an implementation of the "STREAM" benchmark Add "a[i]=b[i]+c[i]" is considered. In each thread, the arrays “a” and “b” are added and the result is stored in the array c. Here is the code that will be executed in each of n (n is variable) threads in parallel:

```
//intel compiler pragmas
#pragma vector nontemporal
#pragma unroll(8)
for(ii=0; ii<length; ii++)
  aa[ii]=bb[ii]+cc[ii];
```

The array length is selected to ensure that the data will not fit in cache (length=2^{25}). The test is repeated ten times. The Intel Compiler *"nontemporal"* pragma turns on the data write protocol, which is called "non-temporal" or "streaming". Compared to a normal write operation, the time for data storage and the space in the cache are saved, because the elements of the "c" array are not loaded into the cache before the data is written in the memory modules. A corresponding pragma for the GNU compiler is not known to the author.

In this deliverable, we consider two cases that are distinguished by different pinning strategy of the threads to the cores. The core id "core_id" is calculated for the thread with number “thread_num” with a formula: coreid=thread_num*core_dist. In the first pinning strategy, the distance between cores is set to 1. In the second strategy, the distance between cores is set to "num_cores/num_threads", where "num_cores" is the total number of cores (64) and "num_threads" is the number of active cores.

[7] We consider the metrics KB, MB, GB as the “true” size of data: 1KB=2^{10}B, 1MB=2^{20}B, 1GB=2^{30} B

Figure 46 shows the bandwidth of the benchmark. Several tests were performed. The variations in performance and power measurements were less than 2 percent. The variations in performance measurements between Excess and Hawk are also in this range.

On the left side, the threads were pinned with the first pinning strategy, where the distance between the threads is one. On the right side, the threads were pinned with the second pinning strategy, where the threads were spread across the cores of the process with a distance equal to "num_cores/num_threads". It is also clear to see how the partitioning of the processor into four NUMA domains (labeled as NPS in Figure 44) distributes the resources of the memory channels between the cores. This is the reason why the highest bandwidth was archived already with the eight cores (179 GB/s). The optimal threads distribution in terms of performance for this algorithm is if each of eight CCDs gets a thread.

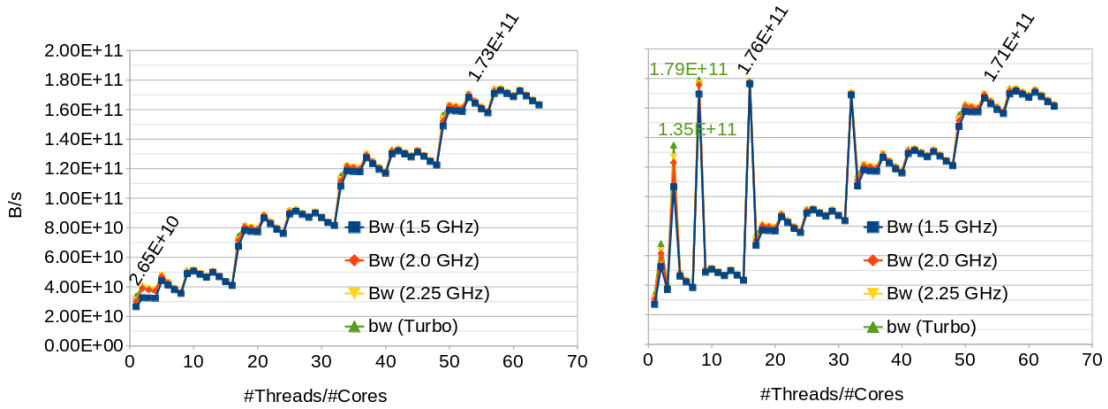

Figure 46: Bandwidth of the benchmark a[i]=b[i]+c[i] on node01 in case of data in the main memory and two pinning strategies with core_dist=1 and core_dist = num_cores/num_threads

Figure 47 shows the electric power during the benchmark tests, measured with the external measurements. It can be easily seen that the turbo mode needs significantly more electrical power, which requires a significant amount of additional energy. This is also seen in Figure 48 and Figure 49. It can be also seen that the turbo needs significantly more electrical power, which requires a significant amount of additional energy.

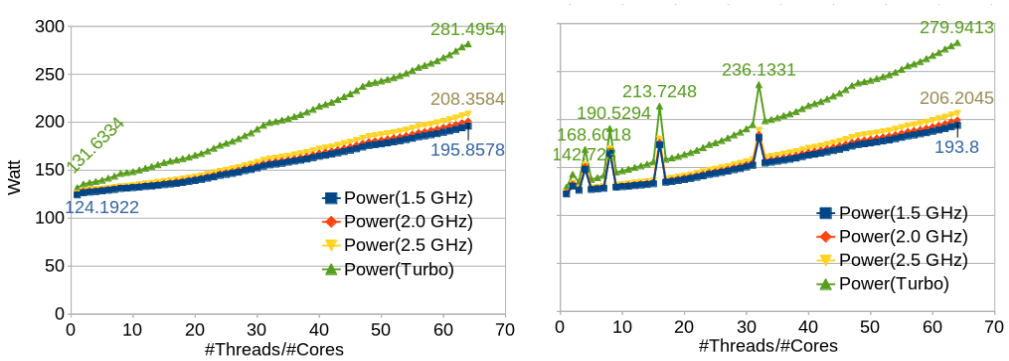

Figure 47: Power consumption of the benchmark a[i]=b[i]+c[i] on node01 in case of data in the main memory and two pinning strategies with core_dist=1 and core_dist = num_cores/num_threads

Figure 48 shows the energy consumption per one byte that is transported between cores and the memory. The energy is calculated with the simple formula "(Power+50 W)/Bandwidth = Joule/Byte". The diagrams show how it is important to find the optimal configuration of both the application and the processor in order to use the hardware efficiently. For example, when running a stream algorithm, which is memory bound, on the Hawk, one can save up to 37% energy with optimal configuration, because at least until now there is no possibility to adjust the CPU frequency on the Hawk[8].

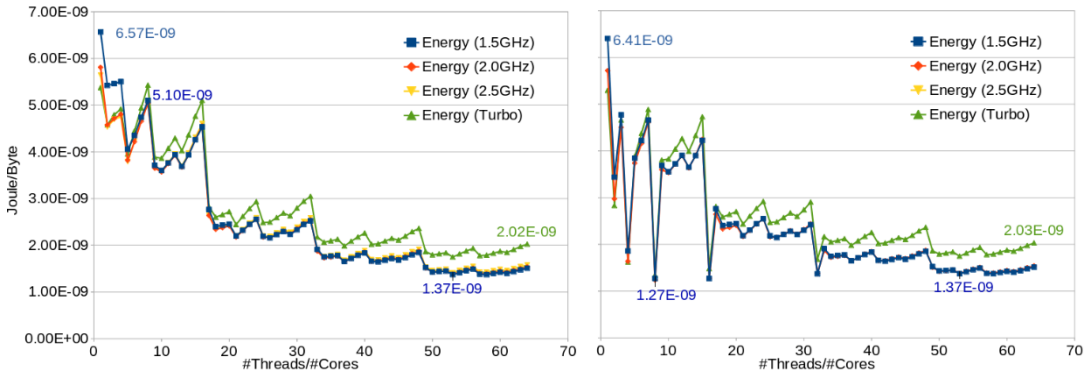

Figure 48: Energy consumption of the benchmark a[i]=b[i]+c[i] on node01 in case of data in the main memory and two pinning strategies with core_dist=1 and core_dist = num_cores/num_threads

5.3.4 Conclusion and Outlook

The initial results show that the Excess cluster and its measurement system are well suited for the objectives of Task 5.4. Although, we still need to work on the details to be able to transfer the results to the supercomputer Hawk. For example, the question of the energy consumption of the cooling system and the power consumption of the processors during the exchange of messages between the nodes should be clarified. This also includes the MPIIO phases, where the data are written via the high performance network into the Lustre file system. Lustre's file system architecture allows parallel access to the multiple raids of the hard disks in order to read or write the files by multiple clients in parallel. The raids are grouped together by Object Storage Targets (OSTs), which are worked independently from each another and connected to high performance network via Object Storage Servers (OSS). Hence, all OSTs can be accessed from every computing node of the supercomputer as parallel as possible.

The two benchmarks show two boundary cases for the compute and memory bound codes. The real applications mainly consist of the different phases with the different $Byte/Flop$ rates and thus have different optimal configurations. For example, compute bound code has the best efficiency if all cores are active. In contrast, the memory bound codes need fewer cores to show better energy efficiency. The second of the considered pinning strategy,

[8] There are certain technical reasons for this. Besides, a wrong choose of CPU frequency can even increase the energy consumption.

where the distance depends on the number of active cores, is optimal for both cases. Therefore, it is beneficial if the HPC services are written using Hybrid Paradigm programming model, such as OpenMP/MPI (see D 4.1 and D4.2).

Another observation is that the current BIOS settings cause the processor to have a higher consumption even when no computing task is performed. This is also confirmed by the data of the six sensors of the computing node "node01", which can be seen on the diagram in Figure 49. No user application was running during the measurement. The small spikes were most likely caused by services of the operating system. The data reordered at the shunts (current sensor) was filtered with a median filter and the voltage signals with a mean filter. The average power consumption is 114.6 W. For comparison, the power consumption of a Haswell processor E5-2680 v3 with 64 GB memory[9] consumes without a computational task an average of 18.2 W.

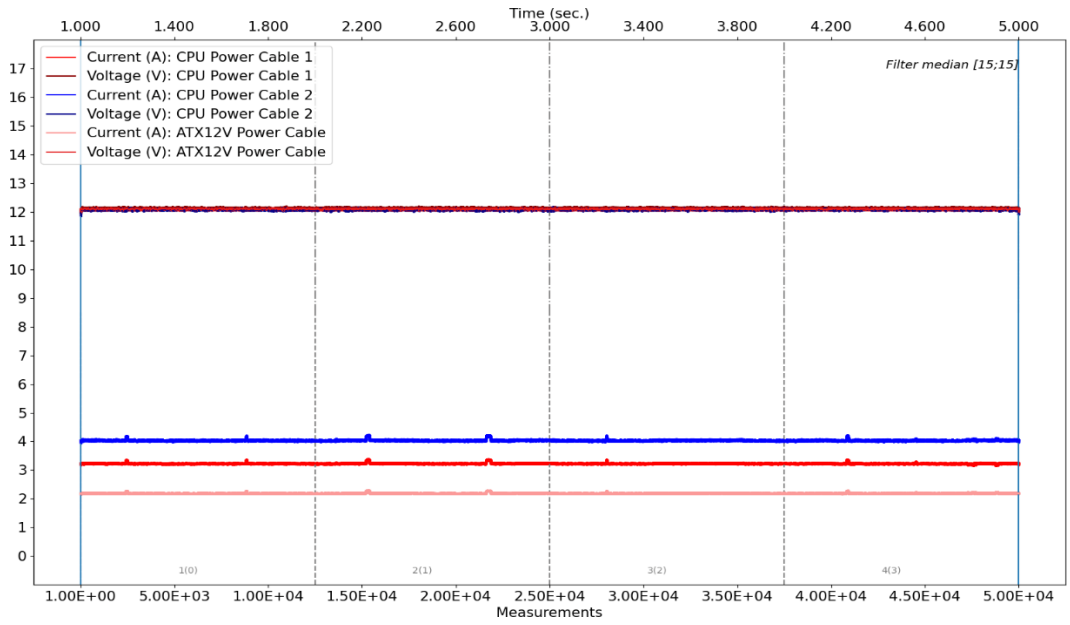

Figure 49: Current (A) and Voltage (V) at six sensors integrated in node01 of Excess cluster

If the data of both processors, Haswell and Rome, is compared, then one doesn't see a significant advance in terms of energy efficiency in case without the computational load, even though the AMD processor was introduced to the market 5 years after the Haswell processor. For this reason, it is relevant to check the other BIOS settings to see how these affect the power consumption without computational load on the one hand and the performance during the computation on the other.

[9] The Haswell processor E5-2680 v3 was introduced to the market by Intel in 2014. A Haswell E5-2680 v3 has 12 processor cores, 24 HW threads, 30 MB L3 cache, 256 kB L2 cache and 32 kB L1 data cache and has four DDR4 memory channels that can be clocked with 2133 MHz.

6 Service Assurance and Remediation

One of the most important considerations when dealing with service assurance and potential autonomous/intelligent remediation is the detection of unwanted behavior of any monitored system, application or service. To this end we will extend and integrate a tool developed by the UVT team.

6.1 Event Detection Engine

A crucial step for service assurance and remediation is the timely detection of any performance, behavior and time/sequence related anomalies. The Event Detection Engine (EDE) will serve as the main service which will be used to detect any such complex anomalous events.

When dealing with distributed systems deployed on heterogeneous hardware platforms it is not a question of if but rather when anomalous events will occur. In order to initiate autonomous remediation of any such events we must first have a reliable anomaly detection mechanism. While simple point anomalies are quite easy to detect with relatively simple rule based mechanisms, contextual and/or sequential anomalies of multivariate data is a challenging issue. EDE provides a wide set of ML methods which are specifically chosen to enable the detection of just such anomalous events.

In the context of SERRANO, we aim to improve EDE with a few key features and capabilities. The main objectives for SERRANO are:

- Identify ML detection methods which are suitable for the detection of anomalous events in the Serrano context.
- Implement ML predictive model optimization mechanisms which can be used for both predictive performance optimization as well as user defined constraints (i.e. inference times, computational resource utilization, model size etc.).
- Implementation of Explainable AI mechanisms which can give meaningful insight into what caused a particular anomalous event to occur (root cause analysis).
- Integration into the Serrano toolchain, this is especially true for how EDE should report detected anomalies.

Figure 50 shows the overall architecture of EDE. It is comprised of 5 main components, each tasked with a specific functionality necessary for both creating and exploiting ML based detection methods. It is implemented using the Python programming language and uses Dask [53] as an execution backend.

The first component is the ***Data Ingestion*** which ensures that data is available to the ML methods. As of writing this section EDE supports querying from Prometheus [54] and ElasticSearch [55] monitoring platforms. In addition, local labeled data is also supported for

supervised method training in CSV and JSON formats. Legacy support for ARFF files is also enabled in the form of a custom conversion script.

Preprocessing is implemented as a separate component and is capable of formatting as well as augmenting both local training and historical monitoring data so that it can be used for predictive model creation. Data normalization and scaling is also handled by this component. We should note that the resulting scaler is also serialized and can/should be used on streamed live monitoring data.

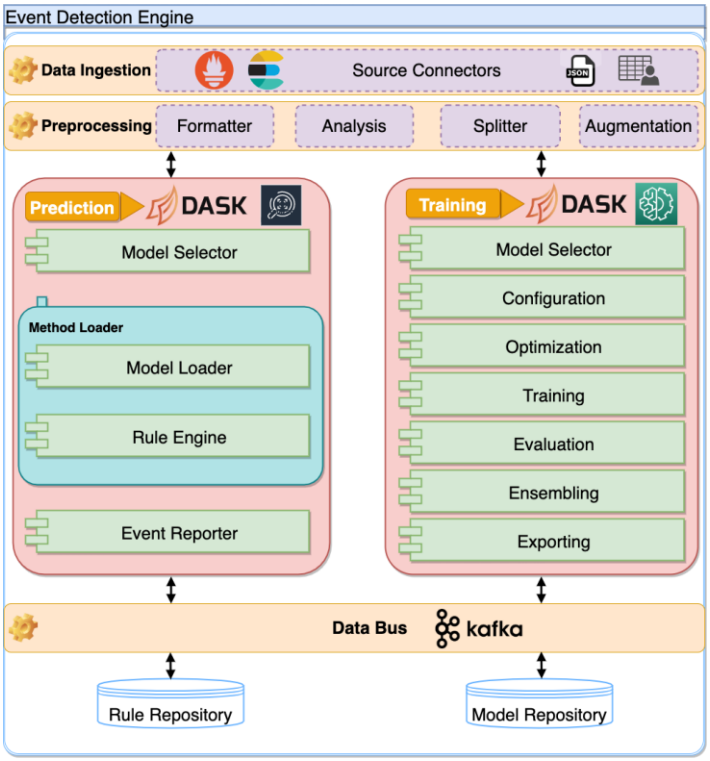

Figure 50: EDE Architecture

In addition, statistical analysis is also executed by the preprocessing component. It is important to note that although there are some predefined data augmentation and analysis methods EDE can execute user defined methods as long as compatibility with the internal EDE processing pipeline is maintained. In essence the processing pipeline uses data frames thus, augmentation and analysis methods need to accept and return Pandas [56] DataFrames.

The ***Training*** component is used to select, configure and optimize ML based predictive models. In case of supervised methods, models need to be trained using a labeled dataset. As with other components within EDE, users can define their own detection methods as long as they are in accordance with the processing pipeline and respect Scikit-learn [57] API naming conventions. Optimization takes several forms. First, we have several Hyper-parameter optimization (HPO) methods ranging from unguided methods such as Grid and Random Search to guided approaches based on genetic algorithms, Bayesian methods, Tree of Parzen Estimators etc. Furthermore, model performance analysis methods and

visualizations such as recursive feature elimination (based on feature importance), training instance selection based on learning curves etc. are also available.

The optimization methods enumerated in the previous sections can be configured via a YAML configuration file. A normal use-case would entail first running HPO on a selected ML method. If a Dask cluster is available each worker will be assigned to evaluate one candidate solution, thus optimization is significantly faster. Once the best performing hyper-parameters have been identified users can define additional optimization analysis methods. All steps from this process are logged and visualization created where applicable. Finally, the predictive models are exported.

Prediction is handled by a separate component. It is capable of instantiating previously serialized methods. All detected anomalies are then forwarded to the EDE data bus based around Kafka [58] topics. This means that other tools and components from Serrano can check this special topic for any anomalous events being detected.

It is clear that simple detection of anomalies is not enough to enable assurance and remediation. An analysis of why a particular event is labeled anomalous is required. To this end we currently support some Explainable AI based methods such as the calculation of Shapley values.

6.2 Detection and Analysis

In this section we will discuss a series of experiments designed for the creation of ML based predictive models for the detection of anomalous behavior. The dataset used for these experiments was created using an anomaly induction tool created by UVT. It is capable of inducing anomalous events which mimic hardware anomalies. In the current dataset we induced 4 anomalous event types:

- **CPU Overload** – Detects the number of physical CPU cores and saturates a user specified amount of cores for a number of seconds. This simulates CPU overloading
- **Memory Eater/Leak** – Writes data into RAM, the amount is specified by the user in KB, MB, GB along with a multiplier and iteration step. This simulates memory interference fault and saturation. It is possible to define also how long this memory allocation is to be maintained.
- **DDOT** – Reputedly calculates the dot product between two matrices. The size of each matrix is calculated based on the CPU L2 cache size reported in the OS. This simulates CPU cache faults. Care should be taken when configuring this type of anomaly as large matrix sizes can cause unpredictable OS behavior causing zombie and/or orphan POSIX processes. Additionally, faulty logging of such events will yield contaminated labeled data.
- **COPY** – Generates and moves a large file from one location to another. Users can set allocation units (KB, MB, GB) and a multiplier. This simulates I/O interference, saturation and failing hard drives. A side effect of this type of anomaly is that it

resembles the Memory Eater/Leak anomaly. This will help us quantify the ability of ML predictive models' capacity in differentiation of the two.

The generated dataset used for these preliminary experiments are generated on a distributed application which is used for clustering of metadata from images uploaded to Flickr which is designed to identify tourist hotspots in Rome Italy.

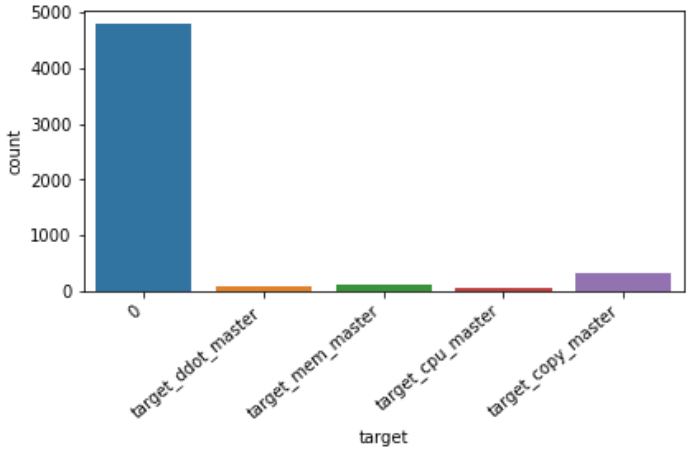

Figure 51: Anomaly distribution in dataset

Figure 51 shows the distribution of event types in the dataset used. There is a total of 5400 events comprised of 90 features. Each feature represents a system level metric collected by Prometheus. We can clearly see that this dataset is extremely unbalanced: 4792 events are normal, 91 are DDOT, 132 are Memory Eater/Leak, 64 are CPU Overload and 321 are COPY.

The next step is to select a classification ML method appropriate for the task of anomaly detection in imbalanced datasets and optimizing its hyper-parameters. For the sake of brevity, we will only represent the result of one ML method although several experiments have been conducted with a total of 10 classification methods. For this deliverable we selected the XGBoost [59] method as this library has been shown to perform well with unbalanced datasets, and our experiments show very promising results at this stage.

The HPO method chosen is based around a genetic algorithm. This genetic algorithm was implemented using the DEAP [60] Python library. The parameters of the HPO method are as follows:

- *Population size*: 40
- *Gene Mutation probability*: 0.2
- *Gene Crossover probability*: 0.5
- *Tournament size*: 4
- *Generation number:* 30

The Hyper-parameters for XGBoost were chosen based on their impact on model performance as related to imbalanced datasets. See the following table for a complete list.

Table 20: XGBoost Hyper-parameters

Parameters	Values	Description
Number of estimators Default: 100 HPO: 500	[10, 50, 100, 200, 300, 500, 1000]	Sets the number of gradient boosted trees. Sometimes called boosting rounds. A low value could negatively impact predictive performance while a too high value increases the risk of poor out of sample performance.
Max Depth Default: 6 HPO: 25	[3, 4, 6, 25, 50]	Maximum depth of a tree. Increasing this value will result in a more complex tree thus more likely to overfit.
Learning Rate Default: 0.3 HPO: 0.15	[0.001, 0.01, 0.05, 0.1, 0.15, 0.2]	Step size shrinkage used in update to prevent overfitting. After each boosting step the weight of new features are shrunk based on the learning rate.
Sub-sample Default: 1.0 HPO: 0.2	[0.2, 0.5, 1.0]	Ratio of training instances. If set to 0.5 random samples half of the available data is used for training. This selection is executed every boosting iteration.
Min-Child Weight Default: 1.0 HPO: 1.0	[1, 2, 5, 6]	Minimum sum of instance weights (hessian) needed in a child. If tree partition steps result in a leaf node with the sum of instance weights less than the value set, the building process will stop further partitioning. A large value will result in a more conservative result.
Gamma: Default: 0.0 HPO: 1.0	[0, 0.1, 1]	Minimum loss reduction required to make further partition on a leaf node. Large values are more conservative.

As a result of the HPO method settings and the defined hyper-parameter space a total of 7560 initial candidate solutions have been generated. Each of these candidate solutions have been cross validated using *StratifiedKFold* (Figure 52) with 4 splits. Scoring was done using the Jaccard Index. The same cross validation type was reused for all subsequent experiments.

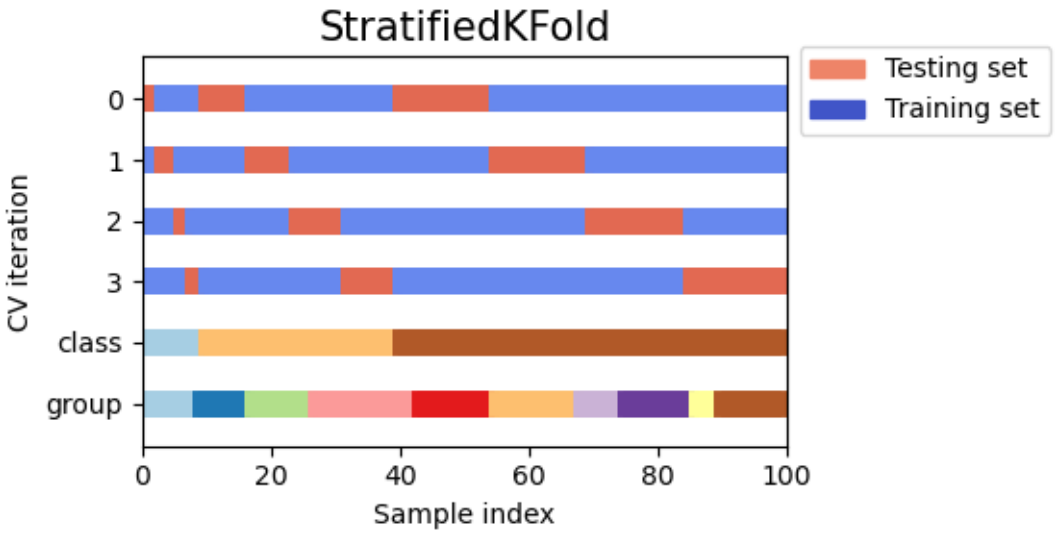

Figure 52: Visualization of StratifiedKFold

Table 21: Classification Report

Classes	precision	recall	F1-score	support
0	0.99729279	0.99937396	0.99833229	4792
ddot	0.95505618	0.93406593	0.94444444	91
memory	0.91366906	0.96212121	0.93726937	132
cpu	0.93548387	0.90625	0.92063492	64
copy	0.99350649	0.95327103	0.97297297	321
accuracy	0.99351852	0.99351852	0.99351852	0.99351852
Macro avg	0.95900168	0.95101643	0.9547308	5400
Weighted avg	0.99357927	0.99351852	0.9935032	5400

HPO yielded hyper-parameter values can be seen in the parameter column from Table 20. A full classification report of a predictive model trained with these parameters can be seen in Table 21. While we can glean some interesting insight from these scores it does not show us a good overview of any misclassifications and what classes are affected by these. For this information we also compute the confusion matrix (Figure 53). We can see that although there are some false positives, all in all the results are inline with what we expected and are promising. Some normal events have been classified as anomalous; the DDOT anomaly being impacted the most. Also, there are 10 instances where Copy anomalies are falsely identified as Memory anomalies. This is mainly due to how Copy generates the files to be written to disk in memory.

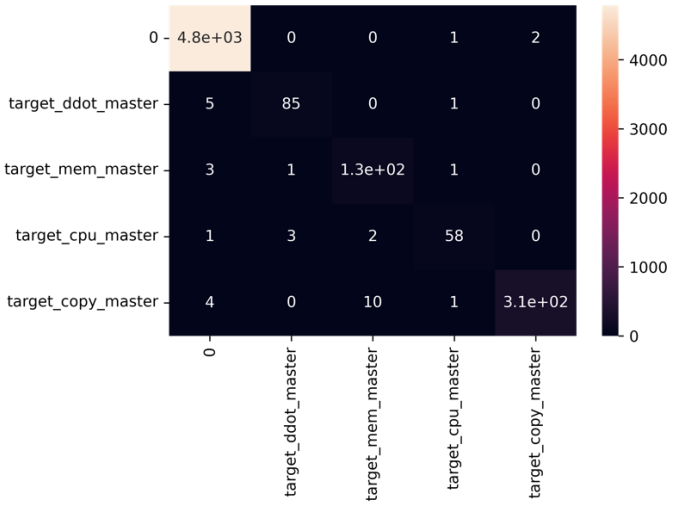

Figure 53: Confusion Matrix Best XGBoost model

Additional model optimization was also carried out, mainly to improve model inference times. First, we gauged the performance of XGBoost on different numbers of training instances. The results can be found in Figure 54. This tells us that for the current dataset if we maintain the distribution of the different anomalous classes we have an acceptable evaluation score starting at just over 2500 events. Which is almost a 100% reduction in the amount of data used for training.

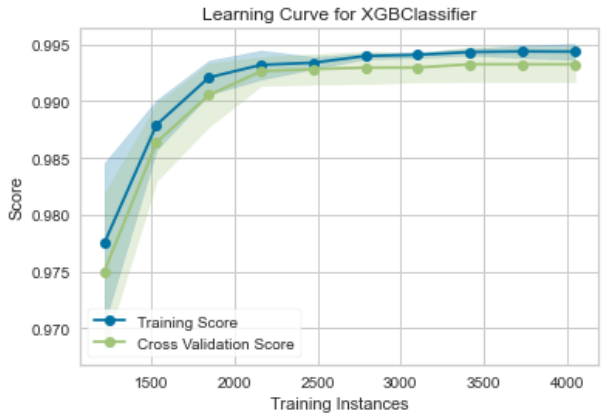

Figure 54: Confusion Matrix Best XGBoost model

Next, we wanted to see if we can further tweak the number of estimators necessary for a viable XGBoost model. This can help in reducing predictive model complexity thus making it easier to instantiate them on limited hardware potentially even Edge/Fog devices.

We know that the best predictive performance was obtained using 500 estimators Figure 55 shows this as well, however we can also see that a lower number of estimators is marked by a sharp decline in predictive performance. This leads us to the conclusion that no acceptable performance can be obtained by reducing the number of estimators, at least not in the case of this dataset. Other avenues of optimization might yield better results.

For example, we can calculate the feature importance resulting after training. This ranking is then used for recursively eliminating features. This way we can sometimes drastically reduce the data necessary for predictive models. Figure 56 shows the result of this analysis. We can see that a relatively small number of features have a significant impact on predictive performance. Although the best performing model was trained using under 20 features, acceptable performance can be obtained from as little as 10 features.

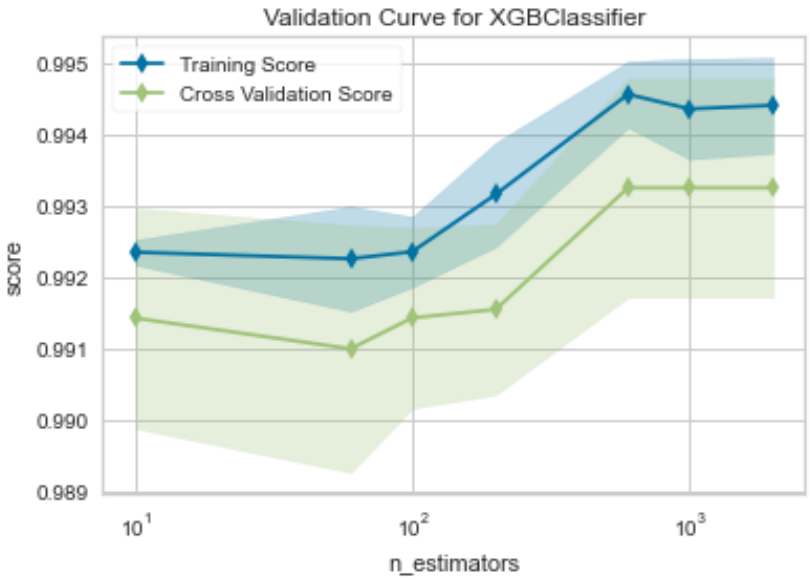

Figure 55: Performance with different number of estimators

6.3 Discussion

In this section we presented the overall architecture of EDE. It will be used as part of the service assurance and remediation mechanisms in SERRANO. The work presented here is just a fraction of the experimental work done. A total of 10 ML methods has been compared and optimized with the same methodologies described here. These results will serve as the basis of an upcoming journal paper.

The next steps will be to use toy applications and initial versions of SERRANO use-cases directly for the generation of new training data. Furthermore, additional metrics (such as the ones from Kubernetes) will be added potentially having a great impact on the types of anomalous behaviour which can be detected.

These initial experiments detailed here use some of the new EDE features which have been added specially for SERRANO, however not all are represented in this chapter. At this time, we have a working initial Explainable AI mechanism in place based on Shapley values. However, as computing Shapley values is extremely costly, approximations must be utilized. Even so, the time it takes for computing is prohibitive, especially for large scale experiments such as the ones detailed here.

While Shapley value is useful for supervised methods their true potential is only apparent when applied to unsupervised anomaly detection methods. An added advantage is that Shapley values are in general easier to compute in the case of unsupervised methods, as inference times are smaller and the number of detected anomalous events is also reduced.

EDE is capable of using several unsupervised methods (via PyOD [61] integration), no experiment has been done at this time. Future work will also have a significant unsupervised component.

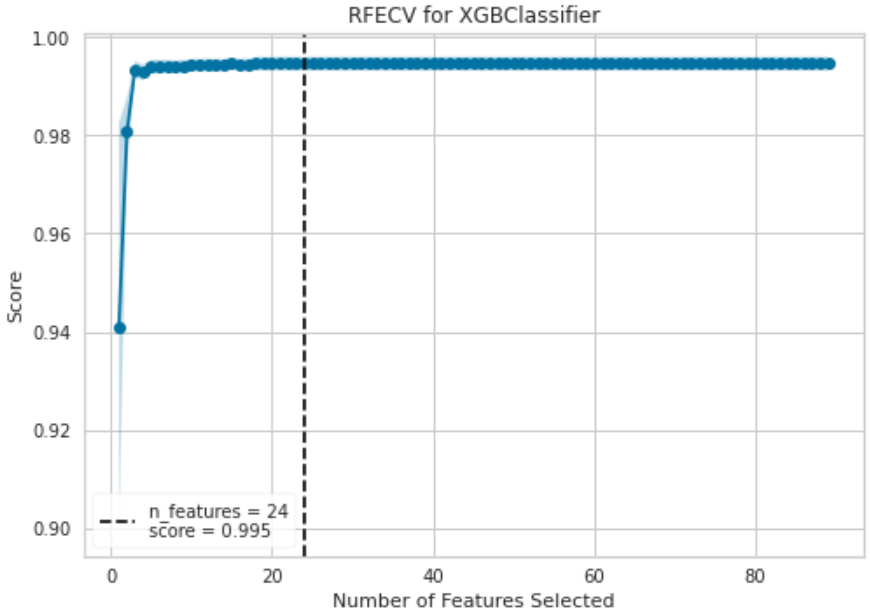

Figure 56: Feature elimination results

7 Conclusions

In this deliverable, we presented a number of resource allocation algorithms that the SERRANO platform will utilize through the Resource Optimization Toolkit (ROT). The algorithms' were described in details and simulation experiments were performed that illustrated their benefits in terms of the targeted objectives. The results indicate that the joint utilization of edge and cloud resources through the realization of the edge-cloud continuum can improve resources and applications efficiency.

ROT provides to the Resource Orchestrator the required optimization logic to its application deployment and service assurance functionalities. ROT has been designed to provide easy addition of new algorithms, fast execution and efficient resource usage while scaling with the execution requests sent by the Resource Orchestrator. ROT integration with the Resource Orchestrator is achieved through two main North Bound Interfaces (NBIs): (i) based on REST and (ii) on the Advanced Message Queuing Protocol (AMQP).

Also, a number of benchmarks were executed, showcasing that the Excess cluster and its measurement system are well suited for evaluating the performance and energy efficiency of HPC services. The Excess cluster can be used to serve as a test platform for the SERRANO orchestrator, while being able to transfer the results to the supercomputer Hawk. We also presented the overall architecture of the Event Detection Engine (EDE) part of the service assurance and remediation mechanisms in SERRANO. Simulations results were performed for the creation of ML based predictive models for the detection of anomalous behaviour. The above will be extended in the future with more resource allocation algorithms, evaluations and predictive models, while developments will take place so as to integrate the related components to the SERRANO platform.

8 References

[1] Reza Shokri and Vitaly Shmatikov, “Privacy-Preserving Deep Learning,” ACM SIGSAC Conference on Computer and Communications Security (CCS), 2015.

[2] Jakub Konečný, Brendan McMahan, and Daniel Ramage, “Federated optimization: Distributed optimization beyond the datacenter,” arXiv preprint arXiv:1511.03575 (2015).

[3] T. Mohammed, C. Joe-Wong, R. Babbar and M. D. Francesco, “Distributed Inference Acceleration with Adaptive DNN Partitioning and Offloading,” IEEE INFOCOM 2020.

[4] X. Chen, J. Zhang, B. Lin, Z. Chen, K. Wolter and G. Min, “Energy-Efficient Offloading for DNN-Based Smart IoT Systems in Cloud-Edge Environments,” in IEEE Transactions on Parallel and Distributed Systems, vol. 33, no. 3, pp. 683-697, March 2022.

[5] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al., “Large scale distributed deep networks,” In NIPS, pp. 1223–1231, 2012.

[6] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman, “Project adam: Building an efficient and scalable deep learning training system,” In 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), pp. 571–582, 2014.

[7] P. Mach, and Z. Becvar, “Mobile edge computing: A survey on architecture and computation offloading,” IEEE Communications Surveys & Tutorials, 19(3), 1628-1656, 2017.

[8] H. Li, K. Ota, M. Dong, “Learning IoT in Edge: Deep Learning for the Internet of Things with Edge Computing,” IEEE network 32.1, 2018.

[9] J. Zhang, L. Khaled, “Mobile edge intelligence and computing for the internet of vehicles,” Proceedings of the IEEE 108.2, 2019.

[10] K. Lin, C. Li, Y. Li, C. Savaglio, G. Fortino, “Distributed Learning for Vehicle Routing Decision in Software Defined Internet of Vehicles,” IEEE Transactions on intelligent transportation systems, 22(6), 2021.

[11] X. Zhao, P. Sun, Z. Xu, H. Min and H. Yu, “Fusion of 3D LIDAR and Camera Data for Object Detection in Autonomous Vehicle Applications,” in IEEE Sensors Journal, vol. 20, no. 9, pp. 4901-4913, 2020.

[12] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger and P. B. Gibbons, “PipeDream: Fast and Efficient Pipeline Parallel DNN Training,” arXiv:1806.03377v1, 2018.

[13] W. Hart, et. al.,“Pyomo–Optimization Modeling in Python,” Springer, 2017.

[14] “IBM CPLEX optimization studio,” available online: https://www.ibm.com/products/ilog-cplex-optimization-studio

[15] “Mapping out edge computing: How dense is it?,” available online: https://www.lightreading.com/the-edge/mapping-out-edge-computing-how-dense-is-it/d/d-id/771128.

[16] “Amazon ec2 pricing,” available online: https://aws.amazon.com/ec2/instance-types/p3/

[17] “Edge computing and transmission costs,” available online: https://www.datacenterdynamics.com/en/opinions/edge-computing-and-transmission-costs/

[18] “The economics of edge computing,” available online: https://edgecomputing-news.com/2020/10/29/analysis-economics-of-edge-computing

[19] “Nvidia resnext performance,” available online: https://ngc.nvidia.com/catalog/resources/nvidia:resnext_for_tensorflow/performance

[20] P. Mattson, et al., “MLPerf Training Benchmark,” ArXiv abs/1910.01500 (2020).

[21] “NVIDIA Data Center Deep Learning Product Performance,” available online: https://developer.nvidia.com/deep-learning-performance-training-inference#dl-inference

[22] Flask 2.0: https://flask.palletsprojects.com/en/2.0.x/

[23] Pika: https://pika.readthedocs.io/en/stable/

[24] PyQt: https://riverbankcomputing.com/software/pyqt/intro

[25] IDC and Seagate, “Data age 2025: The evolution of data to life-critical,” 2017.

[26] M. Hadji, “Scalable and cost-efficient algorithms for reliable and distributed cloud storage,” in International Conference on Cloud Computing and Services Science, pp. 15–37, Springer, 2015.

[27] J. Li et al., “Erasure coding for cloud storage systems: A survey,” Tsinghua Science and Technology, 2013.

[28] W. Shi et al., “Edge computing: Vision and challenges,” IEEE internet of things journal, pp. 637–646, 2016.

[29] T. G. Papaioannou et al., “Scalia: An adaptive scheme for efficient multi-cloud storage,” in SC’12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pp. 1–10, IEEE, 2012.

[30] G. Liu et al., “An economical and slo-guaranteed cloud storage service across multiple cloud service providers,” IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 9, pp. 2440–2453, 2017.

[31] Y. Mansouri et al., “Brokering algorithms for optimizing the availability and cost of cloud storage services,” in 2013 IEEE 5th International Conference on Cloud Computing Technology and Science.

[32] Y. Ma et al., “An ensemble of replication and erasure codes for cloud file systems,” in 2013 Proceedings IEEE INFOCOM, pp. 1276–1284, IEEE, 2013.

[33] Q. Zhang et al., “Charm: A cost-efficient multi-cloud data hosting scheme with high availability,” IEEE Transactions on Cloud computing, pp. 372–386, 2015.

[34] Z. Wu et al., “Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services,” in Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pp. 292–308, 2013.

[35] S. Mu et al., “µlibcloud: Providing high available and uniform accessing to multiple cloud storages,” in 2012 ACM/IEEE 13th International Conference on Grid Computing, pp. 201–208, IEEE, 2012.

[36] H. Abu-Libdeh et al., “Racs: a case for cloud storage diversity,” in Proceedings of the 1st ACM symposium on Cloud computing, pp. 229–240, 2010.

[37] P. Wang et al., “An ant colony algorithm-based approach for cost-effective data hosting with high availability in multi-cloud environments,” in 2018 IEEE 15th International Conference on Networking, Sensing and Control (ICNSC), pp. 1–6, IEEE, 2018.

[38] P. Wang et al., “Optimizing data placement for cost effective and high available multi-cloud storage,” Computing and Informatics, vol. 39, no. 1-2, pp. 51–82, 2020.

[39] P. Wang et al., “An adaptive data placement architecture in multicloud environments,” Scientific Programming, vol. 2020, 2020.

[40] D. P. Bertsekas et al., “Rollout algorithms for combinatorial optimization,” Journal of Heuristics, no. 3, 1997.

[41] “Gurobi optimizer.” https://www.gurobi.com/.

[42] AMD, “AMD HPC Tuning Guide for AMD EPYCTM Processor”. http://developer.amd.com/wp-content/resources/56420.pdf

[43] Usman Pirzada. About Intel’s 10nm Process Lead. Accessed: 2019-11-04. Aug. 2018.: https://wccftech.com/analysis- about-intels-10nm-process

[44] die.net: cpupower(1) - Linux man page, https://linux.die.net/man/1/cpupower

[45] Jülich Supercomputing Centre (JSC), JUWELS. https://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUWELS/JUWELS_node.html

[46] The High-Performance Computing Center Stuttgart (HLRS) of the University of Stuttgart, HPE Hawk,: https://kb.hlrs.de/platforms/index.php/HPE_Hawk

[47] NEC SX-Aurora TSUBASA Documentation, https://www.hpc.nec/documentation

[48] NEC Aurora Web Forums, https://www.hpc.nec/forums

[49] Khabi, Dmitry: Energieeffizienz von Prozessoren in High Performance Computinganwendungen der Ingenieurwissenschaften, Chapter „Elektrische Leistung der Rechenknoten“, Page 160, Stuttgart. Universität Stuttgart, Dissertation, 2018. url: https://elib.uni-stuttgart.de/handle/11682/10827

[50] Khabi, Dmitry & Küster, Uwe. (2013). Power Consumption of Kernel Operations. 10.1007/978-3-319-01439-5_3

[51] Khabi, Dmitry: Energieeffizienz von Prozessoren in High Performance Computinganwendungen der Ingenieurwissenschaften, Chapter „Messung in Wechselstrom“, Pages 128- 131, Stuttgart. Universität Stuttgart, Dissertation, 2018. url: https://elib.uni-stuttgart.de/handle/11682/10827

[52] Khabi, Dmitry: Energieeffizienz von Prozessoren in High Performance Computinganwendungen der Ingenieurwissenschaften, Chapter „Kernel-Operation PETsC-CG auf Hazel Hen“ Pages 93- 100, Stuttgart. Universität Stuttgart, Dissertation, 2018. url: https://elib.uni-stuttgart.de/handle/11682/10827

[53] Dask: Scalable analytics in Pyhton: https://dask.org/

[54] Prometheus - Monitoring system & time series database: https://prometheus.io/

[55] Elasticsearch: https://www.elastic.co/elasticsearch/

[56] pandas – Python data analysis library: https://pandas.pydata.org/

[57] scikit-learn - Machine learning in Python: https://scikit-learn.org/stable/

[58] Apache Kafka: https://kafka.apache.org/

[59] eXtreme Gradient Boosting: https://github.com/dmlc/xgboost

[60] DEAP- Distributed Evolutionary Algorithms in Python: https://deap.readthedocs.io/en/master

[61] PyOD - Python Outlier Detection: https://github.com/yzhao062/pyod

[62] K. Kontodimas, P. Soumplis, A. Kretsis, P. Kokkinos and E. Varvarigos, "Secure Distributed Storage on Cloud-Edge Infrastructures," 2021 IEEE 10th International Conference on Cloud Networking (CloudNet), 2021, pp. 127-132, doi: 10.1109/CloudNet53349.2021.9657156.