Autopoietic Cognitive Edge-cloud Services

Deliverable 4.1
Goal representations corresponding to SLAs and SLOs for use in AI-/ML-/ swarm-based methodology realizing autonomy and actionability

Grant Agreement Number: 101093126

Autopoietic Cognitive Edge-cloud Services	
Project full title	Autopoietic Cognitive Edge-cloud Services
Call identifier	HORIZON-CL4-2022-DATA-01
Type of action	RIA
Start date	01/ 01/2023
End date	31/12/2025
Grant agreement no	101093126

D4.1 – Goal representations corresponding to SLAs and SLOs for use in AI-/ML-/ swarm-based methodology realizing autonomy and actionability	
Author(s)	Matjaž B. Jurič, Melanija Vezočnik, Andreas Kercek, Ksenia Harshina, Fernando Ramos, Suresh Patoria, Cláudio Correia, Loris Cannelli
Editor	Melanija Vezočnik
Participating partners	UL, LAKE, INESC-ID, HIRO, IDSIA
Version	3.0
Status	Completed
Deliverable date	M8
Dissemination Lvl	PU - Public
Official date	31 August 2023
Actual date	31 August 2023

Executive Summary

This deliverable includes an overview of the goal-oriented representations and processes for achieving efficient workload placement within ACES. The aim of ACES is to manage the allocation of workloads across its infrastructure intelligently, ensuring the fulfilment of specific business and operational objectives related to the use cases. To this end, this deliverable discusses some key aspects that empower ACES to deliver high-performance, secure, and reliable edge-cloud services. This is a living document and, therefore, subject to changes during project execution. We will issue revisions to this document if necessary.

Disclaimer

The ACES consortium consists of the following partners:

No	PARTNER ORGANISATION NAME	ABBREVIATION	COUNTRY
1	INSTITUTO DE ENGENHARIA DE SISTEMAS E COMPUTADORES, INVESTIGACAO E DESENVOLVIMENTO EM LISBOA	INESC ID	PT
2	HIRO MICRODATACENTERS B.V	HIRO	NL
3	TECHNISCHE UNIVERSITAT DARMSTADT	TUD	DE
4	LAKESIDE LABS GMBH	LAKE	AT
5	UNIVERZA V LJUBLJANI	UL	SI
6	UNIVERSIDAD POLITECNICA DE MADRID	UPM	ES
7	MARTEL GMBH	MAR	CH
8	SCUOLA UNIVERSITARIA PROFESSIONALE DELLA SVIZZERA ITALIANA	IDSIA	CH
9	INDIPENDENT POWER TRANSMISSION OPERATOR SA	IPTO	EL
10	DATAPOWER SRL	DP	IT
11	SIXSQ SA	SIXSQ	CH

Document Revision History

DATE	VERSION	DESCRIPTION	CONTRIBUTIONS
24/03/2023	1.0	Table of contents	UL
19/06/2023	2.0	First draft	LAKE, INESC, IDSIA, HIRO, UL
28/07/2023	2.1	Section 2.3 updated	INESC
30/07/2023	2.2	Some content added	UL
31/07/2023 – 16/08/2023	2.3	Sections 3.2 and 4 updated	LAKE, HIRO
18/08/2023 – 21/08/2023	2.4	Sections 2.4 and 6 added, Introduction updated	UL, LAKE
22/08/2023 – 30/08/2023	2.5	First review	Félix Cuadrado (UPM), Thien Duc Nguyen (TUDa), Giovanni Rimasa (MAR), Markus Miettinen (TUDa), Fred Buining (HIRO), Ritesh Kumar (HIRO)
31/08/2023	2.6	Content updated	UL, LAKE, HIRO
31/08/2023	3.0	Final	UL

Authors

AUTHOR	PARTNER
Matjaž B. Jurič	UL
Melanija Vezočnik	UL
Andreas Kercek	LAKE
Ksenia Harshina	LAKE
Fernando Ramos	INESC-ID
Suresh Patoria	HIRO
Cláudio Correia	INESC-ID
Loris Canelli	IDSIA
Ritesh Kumar	HIRO

Reviewers

NAME	ORGANISATION
Duc Thien Nguyen	TUDa
Felix Cuadrado	UPM
Giovanni Rimasa	MAR
Markus Miettinen	TUDa
Fred Buining /Ritesh Kumar	HIRO

List of terms and abbreviations

ABBREVIATION	DESCRIPTION
ACES	Autopoiesis Cognitive Edge-cloud Services
AI	Artificial Intelligence
API	Application Programming Interface
CPU	Central Processing Unit
CRD	Custom Resource Definition
CSI	Container Storage Interface
DLRA	Distributed Long Running Application
DNS	Domain Name System
EMDC	Edge Micro Data Centres
FL	Federated Learning
GPS	Global Positioning System
GPU	Graphics Processing Unit
HCL	HashiCorp Configuration Language
HTTP	Hypertext Transfer Protocol
IDS	Intrusion Detection System
LRA	Long Running Applications
I/O	Input/Output
IOPS	Input/Output Operations per Second
IP	Internet Protocol
JSON	JavaScript Object Notation

KPI	Key Performance Indicator
ML	Machine Learning
NF	Network Function
NFS	Network File System
NVMe	Non-volatile Memory Express
PromQL	Prometheus Query Language
PV	Persistent Volume
PVC	Persistent Volume Claims
QoS	Quality of Service
RAM	Random Access Memory
R/W	Read/Write
SLA	Service Level Agreement
SLI	Service Level Indicator
SLO	Service Level Objective
WL	Workload
WP	Work Package
XAI	Explainable Artificial Intelligence
XML	Extensible Markup Language

Table of contents

1 Introduction

In the dynamic landscape of cloud and edge computing, optimizing workload placement is crucial for meeting performance targets and ensuring efficient resource utilization. This document provides an in-depth overview of the goal-oriented representations corresponding to Service Level Agreements (SLAs) and Service Level Objectives (SLOs) for use in AI-/ML-/ swarm-based methodology realizing autonomy and actionability. By aligning with SLAs and SLOs, ACES aims to intelligently manage the allocation of workloads across its infrastructure, ensuring the fulfilment of specific business and operational objectives related to the use cases.

The following sections delineate the essential components and processes for achieving efficient workload placement configuration creation and maintenance within ACES. From identifying relevant data sources to employing advanced knowledge representation for artificial intelligence (AI), machine learning (ML) and swarm decision-making, this document covers some key aspects that empower ACES to deliver high-performance, secure, and reliable edge-cloud services. These aspects will facilitate the deployment of ACES as a robust and intelligent edge-cloud execution system capable of meeting diverse and demanding workload needs.

The rest of the document is structured as follows. Section 2 delves into the foundational aspects necessary for a successful workload placement strategy, such as identifying relevant configuration sources, establishing SLA and SLO definitions, and addressing various non-functional aspects. Section 3 focuses on the data collection and monitoring aspects necessary to inform the workload placement decisions. This involves capturing and analysing configuration data, run-time metrics, and real-time monitoring of SLA/SLO compliance to facilitate informed decision-making. Section 4 introduces the knowledge representation for AI, ML, and swarm, whereas Section 5 provides specific actions and strategies for workload placement, cache management, and storage allocation. Finally, Section 6 concludes this document.

This is a living document and therefore subject to changes during project execution. We will issue revisions of this document if necessary.

2 Requirements for Goal-oriented Workload Placement Configuration Creation and Maintenance

In the scope of ACES, effective workload placement entails not only identifying suitable resources for the execution but also aligning these decisions with overarching SLAs, SLOs, and performance metrics. This section delves into the pivotal domain of configuring and maintaining workload placement in edge cloud environments. Unlike traditional cloud setups, edge clouds bring forth a unique set of challenges, including variable network conditions as well as resource limitations. Consequently, creating and managing configurations for workload placement requires a meticulous understanding of both the application requirements and the underlying edge infrastructure capabilities. To pave the way for goal-oriented workload placement configuration and maintenance within ACES, it is imperative to establish a comprehensive set of requirements that accommodate the intricacies of edge-cloud environments. They are discussed next.

2.1 Configuration Parameters Characterization

We are going to define Configuration/Parameters for each service/Job. Below is the configuration for service or work:

- Service/ job name
- Objective of each service/job
- Priority of each service/job
- Timeout for each service/job
- Order of execution/completion
- Scheduling of each service/job

2.2 Relevant SLA/SLO Definitions

The **SLA (Service Level Agreement)** is defined as “on time scheduling” of the services which are agreed between management model of load sharing system and load sharing users.

Typically, SLA policy defines how many scheduled jobs should be controlled by a single SLA policy within defined period of time and what and all requested parameters/configuration are required for those work.

An ideal SLA has the following characteristics:

- List of services to be agreed upon
- Performance benchmark of each service
- Tracking of each service
- Monitoring of each service
- Evaluation or reporting of each service
- Diagnostics and error reporting

The SLA defines the workload and user agreement for Jobs and services. SLA defines the goal of the services.
The goal could be defined into two categories: 1. Mandatory/ Primary /Guaranteed goal 2. Non-Guaranteed /secondary goal.

An SLO (Service Level Objective) is a goal which is supposed to be achieved by a SLI (Service Level Indicator, SLI is a metric that indicates, measure of performance at a defined time) over a defined period of time.

An ideal SLO has the following characteristics:

- Evaluation Period, the defined time period over which a SLI is to be evaluated.
- Performance Threshold, where the threshold for performance of jobs over the evaluation period is specified.
- Performance graph, which displays a chart that shows the performance threshold and graph that shows the results of evaluating the SLI over the evaluation period.

Below are **SLO (Service Level Objective)**.

- **Maximum Workload Objective:** The maximum list of workloads to be defined.
- **Workload deadline Objective:** The timeout of each jobs or services defined.
- **Concurrent execution Objective:** The list of concurrent jobs and services to be defined.
- **Performance Objective:** The work or services execution/completion per hours are defined.

2.3 Non-functional Aspects

Several non-functional aspects of ACES are crucial for ensuring its proper behaviour: performance, efficiency, security, and robustness. These aspects aim to enhance performance to meet SLA requirements and achieve predefined efficiency goals. Security and robustness considerations are also crucial to ensure resilience against threats targeting the distributed ACES system.

One critical aspect is **ACES performance**, as meeting SLA latencies is crucial, particularly for real-time workloads. We will leverage several forms of network function (NF) acceleration to achieve low latencies and high throughputs at the edge by running NFs in programmable networking hardware (such as Intel Tofino). We plan to leverage in-network analysis, processing, and telemetry to facilitate real-time data and metrics collection, thereby improving the speed of ML training, inference, and cryptographic operations. As it is a challenge to build accelerated code, given both the variety of accelerator targets and the intricacies of acceleration techniques (e.g., how to parallelize code without breaking its semantics?), we will explore program synthesis to meet this challenge.

A second important non-functional aspect is ML retraining **efficiency**. It is essential to consider the computational cost of retraining on the edge nodes, particularly. A self-adaptation ML mechanism integrated into ACES should estimate the expected performance improvement from retraining. Based on this estimation, one can make informed decisions regarding the necessity of retraining or the potential relocation of ML tasks to other nodes within the system.

Robustness is another crucial aspect of ensuring uninterrupted availability of workload state and data, even in the face of node failures. The fault tolerance mechanisms in ACES must strike a balance between secure replica localization (geographically distant) and proximity to sources/clients (requiring topology information). Making calculated decisions regarding the replication factor and replica location is imperative to minimize the latency impact during failure recovery. Employing auditing techniques can further validate the correct location and integrity of state replicas at the edge.

We will also develop **robust and secure ML-based mechanisms** for network control, namely for intrusion and anomaly detection, network telemetry, and traffic adaptation and optimization. We will investigate primitives for AI-based self-monitoring to detect malicious attacks and self-driving network control on various cognitive layers. We will also consider defences against adversarial inputs to AI-based network control.

Also paramount in ACES **security** is maintaining the privacy and integrity of sensitive data and workloads. Considering the exposed location of edge nodes, robust security measures are necessary. ACES should enforce security policies to determine the execution of sensitive data or workloads in more secure environments. Trust execution environments can support the execution of privacy-preserving workloads and enforce anonymous authentication. Additionally, implementing secure logging of all local actions at the nodes and leveraging security primitives such as Intrusion Detection Systems (IDS) and AI-based self-monitoring is crucial for the timely detection and mitigation of malicious behaviour.

Addressing these non-functional requirements is essential for ACES to achieve high performance, reliability, and security levels while meeting the requirements set for the system and its users.

2.4 Global and Local Optimization Goals

In the context of workload placement and resource allocation within cloud and edge computing environments, **global optimization goals** refer to the overarching objectives that guide the decision-making processes to achieve the best possible outcomes for performance, resource utilization, and other relevant aspects on a global scale. These goals take into consideration the entire infrastructure and the interconnected components rather than focusing on local optimizations. Global optimization goals that have the potential to be considered in addition to SLA/SLO definitions within ACES encompass the following aspects: performance enhancement, resource utilization, energy efficiency, load balancing, fault tolerance, scalability, dynamic adaptation, and cost optimization.

Performance enhancement would optimize workload placement to maximize the overall system performance by reducing latency, improving response times, and increasing throughput. Optimizing resource utilization would ensure that computational, storage, and networking resources are efficiently used. Energy consumption could be minimized by intelligently distributing workloads to lower power consumption, thus contributing to environmental sustainability and reduced costs. Load balancing would ensure a balanced distribution of workloads across the infrastructure to prevent the overloading of specific nodes, which could lead to performance degradation and resource contention. Fault tolerance would enhance system resilience by minimizing the impact of hardware failures, network disruptions, and other unforeseen events. When designing workload placement strategies, it is of utmost importance to consider scalability in order to ensure that increasing demands are handled without sacrificing performance. Dynamic adaptation would be realized by implementing means of autopoiesis. Hereby, autopoiesis represents the capability of the system to produce more of its own organization principles than the ones produced by analysing the environment (cognition). This would result in regulating workload placements in response to changing conditions, priority, workload patterns, or resource availability. The aspect of cost optimization would deal with minimizing operational costs by considering several factors, such as infrastructure maintenance and hardware provisioning.

Unlike global optimization goals, **local optimization goals** aim to improve a limited area of the system. For example, when considering hardware topology within ACES, local topology means running on the same Edge Cloud Micro Data Centre (EMDC). Similarly, local SLA topology means belonging to the same guarantee level, such as hard real-time, whereas local trust topology means belonging to the same customer and, therefore, crossing no trust boundaries. For these reasons, balancing both local and global goals is crucial to prevent obtaining suboptimal results in the scope of ACES. Local optimization goals that have the potential to be considered in ACES are latency, resource utilization, energy efficiency, load balancing, availability, data location, content distribution, fault tolerance, and scalability.

Here, latency reduction would consider minimizing the time it takes for data or requests to travel between nodes or devices. This could involve placing workloads closer to users or data sources to achieve faster response times. For example, one optimization goal derived from the use cases could be the requirement that the communication latency between the various locations should be below a specified amount of time so that near real-time simulations could be executed. Optimizing local resource utilization might involve ensuring that a particular machine's available CPU, memory, and storage are efficiently utilized to avoid bottlenecks. Energy efficiency at the local level could focus on reducing energy consumption for a single server or device by adjusting power states or allocating workloads to lower-power modes during periods of low demand. Load balancing at the local level would focus on distributing workloads evenly among resources within a limited area, thus preventing overload and maintaining consistent performance for users or applications. For example, one of the use case requirements is consistent performance. The market clearing algorithm has to periodically produce results. High availability could be achieved by placing redundant instances of critical services within a specific region or zone to mitigate the impact of hardware failures. For example, one goal derived from the use cases could be the requirement of high availability so that the energy market is nearly always operational ensuring continuous execution of the market clearing algorithm and enabling users to monitor health indexes of assets, perform the remote management operation, and remote inspection on assets. Data locality would aim to place the workload in the proximity of the data it needs to access. This aspect would reduce data transfer times and optimize data-intensive operations. Another essential aspect represents content delivery, which would consider optimizing the placement of content, such as cached data, to reduce content delivery times. Local fault tolerance would encompass the configuration of workloads to automatically switch to backup resources or nodes within a specific area in the event of failure. Local scalability would deal with designing workload placement strategies that allow for easy scaling of resources within a limited scope, such as adding more instances of a service on a specific server.

3 Identification of Relevant Data

The previous section discussed some fundamental aspects concerning the requirements for goal-oriented workload placement configuration creation and maintenance. They laid the groundwork for defining SLAs and SLOs, including potential expectations concerning quality and performance. In contrast, this section lists relevant data that could be considered in order to meet the expectations set by SLA and SLO definitions. First, promising configuration parameters and metrics are presented, and later, the aspects of SLA/SLO monitoring are discussed. In essence, monitoring represents the practical implementation of the SLA/SLO definitions and includes data acquisition and analysis.

3.1 Configuration

Configuration encompasses the parameters that determine the behaviour, features, and interactions of the application. One can adapt these parameters without changing the programming code. In ACES, possible configuration parameters are:

- **Storage**: Information on storage requirements can serve as a valuable configuration parameter. This could include details such as disk type (speed), location, size, and other relevant factors.
- **Networking**: Meeting the networking requirements is crucial for the workload to perform its tasks effectively. This configuration parameter could include considerations such as latency, bandwidth, speed, and other relevant factors that contribute to optimal performance.
- **Dependencies**: This configuration parameter provides valuable insights into the communication links between the workload in question and other interconnected workloads. Understanding these dependencies enables the improvement of communication effectiveness.
- **Location**: The location aspect provides crucial information regarding the current or desired placement of the workload. This could include the layer (edge, cloud, edge-cloud), region, zone, edge location, etc.
- **CPU**: Considering CPU as a configuration parameter enables the provision of adequate resources to the workload, such as the required number of cores, clock speed, the correct architecture, etc. This ensures optimal performance and compatibility with the workload's specific processing needs.
- **GPU**: To ensure the workloads can leverage the power and capabilities, GPU can be considered as a configuration parameter. For example, information about the number of GPUs, type, etc., can be considered.
- **Memory**: By considering memory requirements and limits as a configuration parameter, it becomes possible to allocate the appropriate amount of memory, ensure optimal performance and prevent potential memory-related bottlenecks for the workloads.
- **Cost**: Using cost as a configuration parameter enables workloads to prioritize cost reductions. Cost considerations could encompass factors such as hardware costs, energy costs, and environmental impact. By incorporating cost-conscious decision-making, workloads can optimize their resource utilization, minimize expenses, and mitigate environmental effects, all while maintaining compliance with the rest of the required configuration criteria.
- **SLI/SLO**: Utilizing SLIs and SLOs as a configuration parameter provides valuable information regarding the expected performance metrics (for example, platform metrics, such as failed requests per second, request latencies, etc.) for the workload/service. It also offers insights into how closely the current SLIs align with the desired expectations (overachieving or underachieving). This information becomes instrumental in optimizing the workload/service to achieve optimal performance and meet the SLO targets as closely as possible.
- **Logs**: Workload logs can serve as valuable configuration parameters, offering insights into various aspects of the workload's behaviour. Among other things, this includes log levels such as info, warning, error, and so on.
- **Fault tolerance**: Fault tolerance as a configuration parameter could encompass the number of replicas, distribution of the replicas (different nodes, availability zones, regions), etc. This information can be used to achieve the desired level of fault tolerance.

3.2 Run-time Metrics

Assigning incoming demand to the right resources (see sec. 4.1) has different dimensions. Workloads or pods have to be placed to the right compute nodes (workload placement) that have sufficient free computational resources of the required kind (CPU, GPU, etc.), enough cache has to be provided to support processing as well as a place for data to be stored or retrieved for processing. Workload placement, caching and storage assignment are the most important dimensions when it comes to matching of demand and supply (resources) in the sense of section 4.1. For the right match

it is of paramount importance to gather the status of the infrastructure in terms of run-time metrics which could be provided by the Kubernetes environment. These metrics provide the status of the (compute) nodes, available cache and storage etc. In this subsection, we group different run-time metrics relevant for workload placement, caching and storage assignment:

Workload (WL) placement related (pod level):

- Local CPU-/GPU-/etc. load/utilization
- Memory/cache utilization (relative and absolute)
- Metrics related to network links (bandwidth, delay, drop rates, availability, link utilization ration in % of nominally available bandwidth)
- Latency/Response time (processing, caching, storing)
- Anomaly detection (anomalies caused by failures, attacks, etc.)
- Contents of WL/pod description (such as requested CPU, RAM, storage, network properties, latency requirements, sequence and amount of resources needed, Priority, etc.)

Cache related:

- Cache hit rate (% of accesses, in the last x sec), Cache hit/miss ratio, Cache R/W rates or R/W during a certain time, # of R/W per application/pod, average I/O, wait per I/O
- Total no. of requests to allocated cache
- Access frequency of cached element
- Available cache
- Cache occupation in the last x sec.
- Amount of free cache, kind and location
- Size distribution of cached items
- Size of data/items to be cached
- Application response time
- Current/Remaining non-volatile memory express (NVMe) write cycles
- Number of wait cycles
- Process runtime, Estimated runtime of an app, workload duration,
- Process memory footprint
- Time needed for caching a specific piece of data
- Performance of cache (IOPS) on different storage class
- Cache space oversubscription
- % of RAM needed for cache
- Cache size history (time series)
- Historical cache utilization (over last 24h)
- Times of day with least amount of available cache
- Average time of an application staying in cache

Storage related:

- Local Storage capability
- Local (available) storage
- Type (R/W)
- Endpoint latency
- Bandwidth usage
- Origin location (IP and GPS)
- Connected local nodes (IP Addresses)
- # of failures in local nodes
- # of occupied data
- # of network disconnections in local nodes
- # and latency to data replicas
- App data usage metrics
- Data policies

During development and implementation of the AI-methodologies (see below) we will focus on run-time metrics from the list above that turn out to be relevant for the optimization processes.

3.3 SLA/SLO Monitoring

SLA and SLO monitoring present critical aspects of ensuring the reliability and performance of services or systems. It is closely connected to the definition of SLAs and SLOs, as the monitoring lays the groundwork to ensure the performance and quality expectations in various contexts are met. Relevant aspects are related to the definitions of SLA and SLO in ACES already discussed in one of the previous sections. SLA monitoring involves tracking the quality metrics to ensure compliance with the agreed-upon terms, while SLOs help translate high-level SLA requirements into concrete objectives for monitoring and managing service performance. Relevant aspects of SLA/SLO monitoring are metrics and data collection, alerting and thresholds, real-time monitoring and visualization, performance analysis, trend analysis, and capacity planning. They are discussed next.

Collecting and exposing various metrics in the applications is of utmost importance to gain valuable insights into their performance and behavior, aiding in monitoring and troubleshooting. There are several solutions for the collection of metrics within the applications. Hereby, one can use the KumuluzEE Metrics[1] module within the KumuluzEE[2] framework— an open-source microservices framework for Java applications designed to simplify the development of microservices-based applications. KumuluzEE Metrics is compliant with the MicroProfile[3] and provides support for collecting different system, application, and user-defined metrics and exposing them in different ways. Metrics can be exposed on a URL as a JSON object or in Prometheus format.

There are several categories of metrics that can be collected within the KumuluzEE Metrics module. These metrics are counter, gauge, concurrent gauge, histogram, meter, timer, and simple timer. A counter measures an incremental value in order to count occurrences of events, such as the number of requests. In contrast, a gauge measures a single numerical value with respect to time, such as monitoring the number of jobs in a queue. Concurrent gauge measures a value that can increase or decrease, whereas histogram measures the distribution of values. A meter measures the rate of event occurrences with respect to time, such as monitoring the method calls. It tracks the number of events as well as the average rate. A timer measures the execution time of a particular block of code or a method, whereas a simple timer measures the total amount of time the block of code has spent executing.

As monitoring and collecting metrics present one of the crucial aspects for understanding the system's performance, identifying potential issues, and making data-driven decisions for improvements within the application, it is beneficial to gain valuable insights into the health and behavior of the application. Here, one can use Prometheus[4]—widespread open-source monitoring and alerting toolkit used for collecting and storing metrics data. Prometheus is a versatile tool as it is designed to work with a wide range of systems and services. There are several ways to collect metrics in Prometheus. For example, Prometheus offers official client libraries for several programming languages. Other approaches include exposition via HTTP endpoints, utilization of the Prometheus exporters, service discovery mechanisms, push gateway, and blackbox exporter. KumuluzEE Metrics supports the integration with Prometheus by means of exposing the metrics via HTTP endpoints by the application. These endpoints serve metrics data in a format Prometheus understands, usually in the Prometheus exposition format. Prometheus scrapes these endpoints at regular intervals to collect the metrics data. Prometheus stores these scraped metrics data in its time-series database. Utilizing Prometheus' built-in querying and visualization features or integration with Grafana enables the creation of dashboards to gain insights into the application's performance. Grafana[5] is an open-source tool for monitoring and analytics. Due to Grafana's support for metrics visualization, it represents an essential component to monitor and observe stacks. It also provides a unified and customizable view of metrics data providing users the basis to make informed decisions, detect anomalies, and troubleshoot problems.

Specific requirements, existing infrastructure, and integration needs present the basis for selecting a tool for expressing and managing SLAs and SLOs effectively in the appropriate format. For representing SLAs and SLOs in JSON and XML formats, the following tools can be considered:

[1] https://github.com/kumuluz/kumuluzee-metrics
[2] https://ee.kumuluz.com/
[3] https://microprofile.io/
[4] https://prometheus.io/
[5] https://grafana.com/

- Prometheus Rules: enable the definition of recording and alerting rules in JSON-like format using PromQL (Prometheus Query Language). This language is designed to express time-series data and can be used to define SLOs based on the metrics collected by Prometheus.
- Grafana JSON Datasource: Grafana provides the option to create JSON-based data sources, where SLAs and SLOs can be defined in JSON format.
- Custom Scripts with JSON/XML Libraries: various programming languages, such as Java and JavaScript, and their respective JSON and XML libraries can be used to create custom scripts that generate SLA/SLO configurations in the desired format. This approach provides more flexibility but requires custom development.
- Configuration Management Tools: tools like Ansible, SaltStack, or Terraform can use JSON or HashiCorp Configuration Language (HCL) to manage infrastructure and configurations, including SLAs and SLOs. These configuration management tools support expressing complex configurations.
- Custom APIs: an application or system that deals with SLAs and SLOs can also use a custom API that accepts JSON or XML representations of these definitions. Then, JSONnet or other tools can be used to generate the necessary configuration.
- XML Templating Libraries: For XML representations, one can use XML templating libraries like Jinja2XML or other XML templating engines to generate XML configurations from templates.

4 Knowledge Representation for AI/ML

In this section we describe methodologies and approaches for creating EMDCs or multiple EMDC that show autopoietic behaviour and how therefore knowledge and goals shall be represented in the ACES framework. We start with a brief description of the ACES overarching architecture in sec. 4.1.1, the components of the ACES architectural framework including agents (demand- and supply agents), etc. In sec. 4.1.2 we present possible representations and algorithms for decision making in the aforementioned agents. These agents will collaborate to achieve global goals as for instance mentioned in sec 2.4. Such goals have to be formulated in particular goal representations which are suggested in sec. 4.2.

4.1 Approaches and Methodologies Used in ACES

The main goal is to consider the EMDC or multiple EMDCs as autopoietic systems facilitated by the architecture developed in WP2. Autopoietic behaviour is desired for functions like cache assignment (caching), storage as well as workload placement/orchestration.

4.1.1 Architecture

Description of the overarching architecture with possible locations and roles of agents:

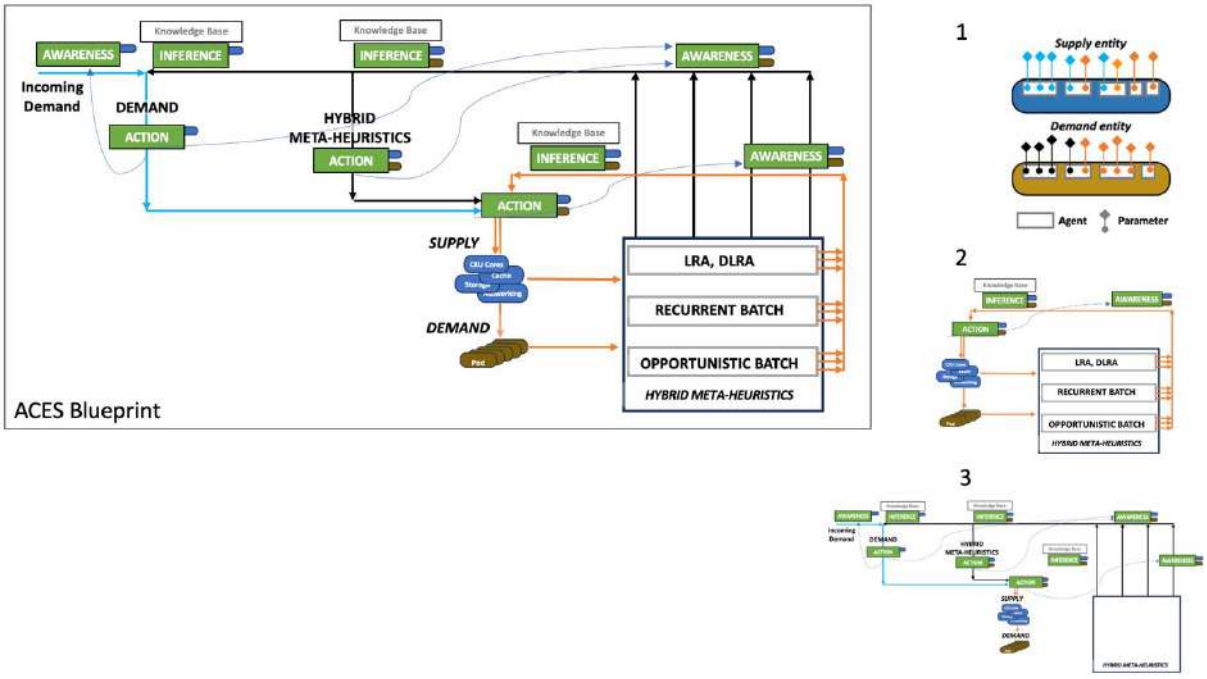

Figure 1: High Level System Architecture

The Optimization logic of hybrid meta-heuristics will build on demand and supply as explained in Figure 1.
Demand and supply will be broken down into entities, representing demand and supply, Demand (user, application, pod) and Supply (CXU cores, Node, Cache, Storage, Networking). Their state and their individual behaviour are characterised with the help of agents and their parameter(s).

- The number, type of agents and their parameter settings are defined based on:
  - Intrinsic behaviours of the entity: For demand those are the intrinsic behaviours for each of three types of workloads, LRA & DLRA, recurrent batch, opportunistic batch. And for supply that is the intrinsic behaviours of CXU cores, Memory, Storage, Networking.
  - Desired behaviour of the entity within the hybrid meta-heuristic of a workload type (LRA & DLRA, recurrent batch, opportunistic batch).
  - Desired behaviour of the entity within the platform as a whole.
- The actual behaviour of entities, groups of entities is continuously monitored (AWARENESS) and analysed (INFERENCE) on two levels:

  - the hybrid meta-heuristic where the entity is deployed in. (See Figure 1, item 2).
  - the platform as a whole (See Figure 1, item 3). In particular:
    - across the intrinsic categories that the entity is part of.
    - across the different hybrid meta-heuristics that are active on the platform.
- The agents of demand and supply entities are auto-configured (ACTION) to optimize the utilization of the platform, match Service Level Agreements (SLA), Quality of Service (QoS), meet KPI's.
- Each type of the three identified workloads (LRA & DLRA, recurrent batch, opportunistic batch), will have one (or more) hybrid meta-heuristic (s) to choose from. Each hybrid meta-heuristic, when active, operates on a local EMDC and can stretch to neighbouring EMDCs (federation) to the extent that neighbouring EMDCs have created an ad-hoc or enduring federation.
- Each hybrid meta-heuristic will match demand (users, applications, pods) and supply (CXU cores, memory usage, storage, networking) through cycles: Awareness (collected metrics), inference (created intelligence) and actions (changes in behaviour of demand and supply) are dynamically customized.

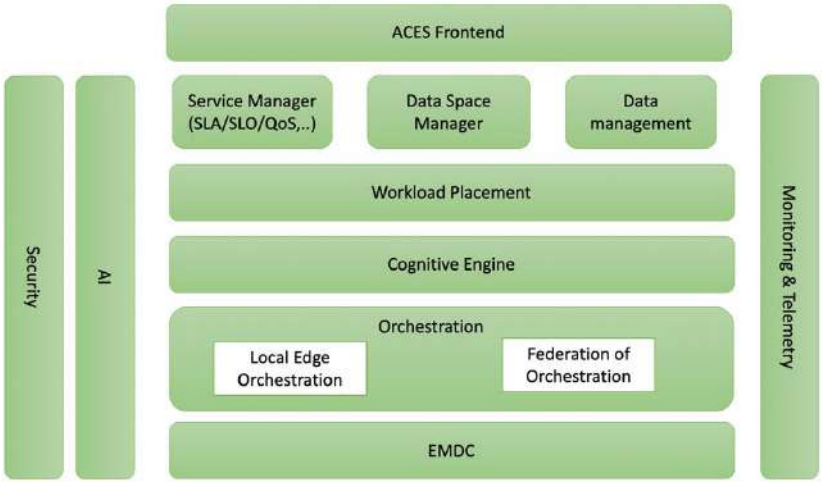

Figure 2: High Stack Architecture

ACES frontend: providing a single panel of glass to end users and operators.

Service manager: group of software components is used to effectively manage the storage, definition, catalogue, and instantiation of workloads (also known as containers). It also manages the QoS for workloads through management of the underlying infrastructure, with supporting components to manage and control EMDC hardware and network infrastructure. Including:

- SLA/SLO
- QoS (Quality of Service)
- Service catalogue
- Data dependency
- Privacy & Security

Date space manager: Providing a secure and privacy-preserving infrastructure to pool, access, share, process and use, and monetize data (data-sets, data-products, AI models). It has a clear and practical structure for access to and use of data, training of AI, all in a fair, transparent, proportionate and/non-discriminatory manner with clear and trustworthy data governance mechanisms. It enforces the European rules and values, in particular personal data protection, consumer protection legislation and competition law. Data space manager through role-based access management will be able to assign and acknowledge a wide variety of roles relevant to operate in data spaces. The Data space will have a store feature that facilitates the market place aspect of the data space, a tool to make data available against compensation, including remuneration, or for free.

Data manager: range of software components to underpin the storage and usage of workload and EMDC data on the platform, monitoring and enforcing the management of data through data policies and lifecycle management.

- DKB (Distributed knowledge base)
- Policy management
- Data lifecycle management

Workload placement: is the positioning of workloads (demand) on resources (supply) within the EMDC platform. The decisions in matching demand and supply are made dynamically and ad hoc through a decentralised process of agents that represent demand and supply and obey meta-heuristic algorithms (incl. swarm technologies) that provide a good enough solution through for example some form of stochastic optimization. Meta-heuristics are lightweight programmable algorithms that solve complex problems within a short execution time and provide a good enough solution. The many independent agents in a meta-heuristic algorithm makes it suitable for parallelization. The use of meta-heuristics has a few disadvantages:

- Communication overhead: in parallel implementations, the communication and data transfer between many agents can be intensive and create an overhead that penalizes the execution of the algorithm and consumes more time. The communication steps require all participants have to stop their individual movements of data between agent and start the communication coherently.
- Lack of explainable behaviour of, and control over, the meta-heuristic: swarm behaviour is emergent group behaviour based on simple rules and agents. Extrapolation from the swarm agents and rules do not necessarily provide a predictable behaviour.

To manage the above mentioned aspects, the meta-heuristic is designed in a way that it creates minimal communication overhead. By adding non-swarm AI/ML optimization algorithms and iterative methods (in the cognitive engine, Cognitive Framework provides the whole ML lifecycle support for different ML models) we create feedback and certain level of control on the swarm behaviour. This will also enable some level of explainability of the swarm behaviour and a mechanism to understand and influence the swarm behaviour in a logical manner. These functions will be implemented in the cognitive engine. Note: This does not mean we will be able to create fully predictable behaviour and control of the swarm emergent behaviour since that would contradict with the concept of emergence.
Cognitive engine: Cognitive framework is the distributed AI, ML training, configuration, prediction, validation and monitoring models for the complete platform and for the autopoietic framework in particular. These AI/ML models will more and more be integrated in the individual hardware modules across the EMDC as CXL becomes more available on hardware component level and implemented across the EMDC platform.

Note: the ACES project will submit a hop-on instrument proposal to implement more CXL capabilities in the EMDC hardware.

Orchestration: Realizes two levels of orchestration. The first level is tuned to the local custom configuration of the EMDC. It anticipates the future full capabilities of CXL disaggregated infrastructure in which storage, memories and CXU cores can be pooled independently and ad hoc for certain workloads (local orchestration). The second level is the federated orchestration in which local autonomous EMDC's create federations with neighbouring EMDC's in order to cope with local and regional workloads.

AI: AI functions/services will be developed to make ACES platform to make automatic intelligent decisions for self-optimization & auto healing and creating federations with neighbouring EMDC's. To create this neighbouring view, the EMDCs support different forms of federated learning across their regional EMDCs.

Monitoring and Telemetry: The telemetry and monitoring system section represents the components used to collect information to be analysed and processed for improving other software and algorithm functions, and to visually represent telemetry data to an administrator.

- Metrics information & filter
- Metrics analyser
- Metadata database
- Data collector
- Dashboard

Security: will have a range of services to identify anomalies in, and mitigate the poisoning of the data, used by the AI/ ML and swarm technology, and secure the AI algorithms/ models.

4.1.2 Agent Representations

In subsection 4.1.1 we briefly explained the blueprint of the ACES architecture placing agents on the demand and supply side. In this subsection we suggest possible mechanisms and algorithms the decision making of the agents can be based on. We particularly mention mechanisms based on gradient-based tracking, swarm intelligence or even ML-based

decision making in agents. They are considered as candidates that may be extended or also merged in the course of selecting the best decision-making algorithms for particular global and local objectives.

Additionally, we describe the agent representation of the ACES architecture in this subsection. We need to implement ML based agent which can support ML/AI based decision making services for Cognitive Engine.

4.1.2.1 Gradient-based tracking

In distributed optimization, the objective is to solve a global optimization problem by utilizing a network of interconnected agents or computing nodes that collaboratively exchange information. These schemes are particularly useful in large-scale optimization tasks where a centralized approach may be impractical or computationally expensive. Gradient tracking distributed optimization schemes aim to leverage local computations and communication among agents to achieve the global optimum efficiently.

The key idea behind gradient tracking is to maintain an estimate of the global gradient (global gradient tracking is used for distributed nature of the data, and push-sum consensus to handle the imbalance caused by the directed nature of the underlying graph) by leveraging local information. Each agent in the network performs its own local computations, which typically involve evaluating a local objective function and computing the gradient using local data. The agents then exchange information, such as their local gradients or model updates, with their neighbouring agents. By aggregating and incorporating this exchanged information, each agent can update its local estimate of the global gradient.

Gradient tracking distributed optimization schemes offer several advantages. Firstly, they allow for parallel and distributed computations, enabling faster convergence and scalability in large-scale optimization problems. Secondly, these schemes are robust to failures or changes in the network topology, as they rely on local computations and information exchange rather than a centralized coordinator. Additionally, they are well-suited for scenarios where data privacy and security are concerns, as agents only exchange minimal information necessary for gradient estimation.

4.1.2.2 Swarm intelligence

Swarm Intelligence is an approach that creates opportunity for optimal, robust, self-sustaining, simple but optimal solutions for various complex problems. The simplicity of the approach that stems from the local interactions and communication of agents prompts an emergent intelligence of the overall system. The resulting self-organizing system shares some similar qualities with autopoietic systems that the ACES project takes inspiration from.

Swarm agents are characterized by the ability to perform simple actions and interact with their neighborhood - other agents of the swarm. Even though the intelligence of one swarm member is simplistic, the amalgamation of multiple swarm agents creates a more sophisticated higher-level intelligence via emergence. Thus, one of the requirements for potential swarm members is a high quantity of possible swarm members. Beside a local intelligence and an element for communication, an agent should also have the ability of decision-making to be a well-suited swarm candidate (e.g., a computational node with decision-making abilities in an EMDC).

The knowledge that is given to or gathered from the agents, which they use to perform actions with, is context dependent. When engineering a specific swarm algorithm onto a problem of interest, it is up to the researcher(s) to establish the extent of the agent's knowledge. For example, in the case of a manufacturing plant, the agents, which were mapped onto lots (a bunch or group of products that have to processed at the same time and in the same processes), had the knowledge of the type and the sequence steps they need to perform for being considered processed. Along with the problem-specific knowledge, the agents possess algorithm-specific knowledge, which depends on the type(s) of the swarm algorithm(s) that was/were chosen during the design process. The algorithm-specific knowledge can refer to, for example, the hormone/pheromone evaporation rate in the case of the Hormone or Ant Algorithm, respectively. The evaporation rates are among the important parameters in Hormone or Ant Algorithms.

We will look at specific swarm algorithms and how they can potentially be engineered to the ACES scenario. The Hormone algorithm is inspired by the behaviour of the endocrine systems of our bodies. This algorithm has been applied earlier to the manufacturing plant problem with successful results. In the ACES case, we can propose the following mapping for agents:

1. Supply swarm agents (on node level and via a pool of resource, e.g., CPU, RAM, storage, GPU, FPGA, etc.)
2. Demand swarm agents (on pod level)

Exemplary, engineering the hormone algorithm would include following main principles:

1. Demand swarm agents are stochastically attracted by the supply swarm agents that express their capacities with the amount of hormone that they possess.
2. Each node generates a hormone of its resource type. Resources with the same hormone type generate the same hormone.

4.1.2.3 Machine Learning

1. ACES Machine learning requirements summary:

In ACES, machine learning model will provide Explainable Artificial Intelligence (XAI) and transparency of the increasing automation in edge-services platform by operators, software developers and end-users. Explainable Artificial Intelligence (XAI) is an emerging area of research in the field of Artificial Intelligence (AI) and XAI . XAI is a set of method, that allows human users to comprehend and trust the results and output created by ML algorithm.

ACES will infuse autopoiesis and cognition on application level, data centre level, cloud level management using AI for different functionalities such as workload placement, service and resource management, data and policy management. The general aim is to research an evolution of cloud computing, an edge services cloud with hierarchical intelligence, and specifically, autopoiesis and cognitive behaviours, to manage and automate a computing platform, network fabric, storage resources, virtualization, and analytics to increase resilience while managing simultaneous service constraints.

ACES refers to a novel kind of Cloud-to-Edge Peer-2-Peer computing system capable of producing and maintaining itself, and its autonomic computing properties, by self-creating and maintaining its own workload placement principles and goes beyond those principles produced by cognition. The peer-to-peer computing system can be built with decentralized federated learning structures in the data centres.

'Cognition' is defined as the ability to acquire and process information, apply knowledge, and autonomously change the inner circumstances to provide better services in response to the fluctuations in the environment. Because cognition by definition creates intelligent self-adaptation up to the same level of complexity as its analysed environment, ACES needs an additional control approach to protect the system from damage or destruction in situations where its cognitive complexity handling capabilities are exceeded.

'Autopoiesis' means self-producing and maintaining itself, and is defined as the ability to produce more of their own organization principles than the ones produced by analysing the environment (cognition). Such as self-configuration, self-optimization, self-protection and self-healing.

The ML models should provide the learning resource utilization, usage patterns and energy consumption, and discovery and management of resources and data sources, finally, the ML models can provide system self- monitoring, managing, organizing, adapting, healing etc.

2. Machine Learning framework

The machine learning part will provide a common framework for all the different ML models. The framework can provide the training data access, training algorithms, validation methods, prediction methods and provide the monitoring data about the whole learning process. Different APIs interfaces can be developed for each ML model. The framework should be able to be integrated with different ML models with their training algorithms, training data configurations, and prediction methods. Each partner can tailor their machine learning model by customizing the training, validation, and prediction algorithms. They should call the ML framework API to register and configure their models in the framework. Other application services or models can call the framework ML APIs to run the ML models for different purposes: security, workload prediction, self-management, alert prediction etc.

A distributed ML scheme like federated learning (FL) can be used for model deployment, communication and aggregation in the ML framework. FL clients can be deployed in each node level. The ML can learn from the distributed data in its node and does not need to access other node's data. So, keep the data in the local server and improve privacy in this way. The model in each node can do peer-to-peer communication and aggregate by neighbourhood nodes. We can provide the ML configuration APIs, which provides the deployment setup information about how to deploy, when to aggregate, and the number of nodes for deployment.

We need to investigate different ML models and analyse how to create the ML related APIS for them to register and integrate with the framework. Further, we need to define the framework ML APIS for different services, applications, models to use the ML functions.

For all the different methodologies mentioned above (gradient tracking, swarm, ML), we need to define the optimization goals that must be met on the system and eventually on the agent level. In autopoietic systems, the global goals shall appear as emergent behaviour of the interacting agents in place. These goals are derived from application- and infrastructure requirements, non-functional aspects as well as SLAs/SLOs, etc., as mentioned above. These goals must be transferred into particular forms, "goal representations", that can be used to monitor the system as well as to adapt its local agent behaviour accordingly. Goal representations in agent-based modelling may be global objective functions, and local agent-related cost functions, etc. that contain goals and KPIs in one or the other form. They are described in the following section.

4.2 Goal Representation

Goal representations is a particular form of global goal that can be used for monitoring the system as well as accommodate the local agent behaviour accordingly. Sometimes it is sufficient to define global goals that have to be met on system level. It depends on the methodology or algorithms for the agent's local decision making if additional local goals for agents have to be defined as well. E.g. for decision making based on swarm algorithms as described below, global goals are sufficient. The agents themselves do not require intrinsic goals but just follow their rules according to the swarm algorithms. The global goals can be met when the algorithms, parameters and the communication of information are selected and tuned properly. For local decision making based on gradient tracking (see below) a particular agent behaviour can be influenced by a local cost function (goal on agent level). However, these local cost functions must be defined in a way that local agent behaviour and the interaction between agents results in the desired system behaviour meeting global goals.
Goal representations for swarm agents:

- Global level: Target values for global metrics such as overall CPU-, cache- utilization, latency, energy consumption, etc. (KPIs). Such KPIs can be represented via either an objective function containing the KPIs as a weighted sum or the objective function may be a ML-based method proposing particular heuristics (swarm algorithms) or parameters for the agents.
- Agent level: The agents make their decisions without any global knowledge (state on system level, global metrics). Their decision is based on local metrics (e.g., CPU load, free cache, etc.) and information from local neighbours (e.g., neighbours CPU-load, free cache, etc.), where locality can be either geographical, or meant as common characteristic of the agents. Depending on the form of swarm algorithm used, the decision may consist of delivering information that influences the environment or neighbours. This can be done either explicitly ("I take this part of the pod", "I send this pod to my preferred neighbour") or implicitly (e.g., creation of (decaying) hormones or pheromones). In both cases, the agent itself does not have an intrinsic goal, cost-function or objective-function except for exactly following its rules given by swarm algorithms with the corresponding parameters. However, local rules and interactions must be "tuned" that the global goals appear as emergent behaviour of the agent's interaction.

Goal representations for gradient-based tracking agents:

- Global level: Each agent in the framework should possess a local cost function that measures the mismatch between the actual and the desired performances (i.e., the KPIs). The local cost function of each agent will depend on parameters of interests; for example, in ACES it would be reasonable to monitor quantities as: available bandwidth, available storage size, available computational power, etc. These quantities could also be subject to local constraints.
  Each agent individually will try to optimize its own local cost function and, at certain times, will exchange some information with its neighbours
  This overall procedure will allow the agents inside the ACES framework to solve cooperatively global tasks (e.g., workload allocation, memory allocation etc.) by exchanging a limited amount of information between each other.
  The global objective function that the agents will implicitly optimize cooperatively by means of local actions will be a weighted sum/average of the local cost functions.
- Agent level: The agents have no global knowledge, and they gather information from their own sensors/sources and from communication with neighbouring agents. The kind of information to be exchanged will depend on the specific distributed gradient tracking scheme that will be implemented; in any case this information will in

general be related to the local gradient of the local cost functions of each agent. Indeed, the gradient information is sufficient to let other agents understand how the optimization of the others is proceeding.
Finally, the presence of ML/AI tools will be considered: a) at the level of a single agent, to do, for example, forecasting of future needs, based on previous history, thus to update the local KPIs; b) among different layers of the cognitive ACES framework, to share knowledge, through, for instance, federated learning.

5 Actions for Workload Placement/Cache/Storage

Actions for workload placement, cache, and storage are a part of Deliverable D4.2 (due M12) that deals with actions language and library. For this reason, one can find only general aspects and possibilities for workload placement actions, cache, and storage in ACES herein. This section lays the groundwork for preparing the aforementioned deliverable, considering the possible prospects for workload placement actions that include but are not limited to deployment descriptors, node affinity, helm charts, and pod migration. The remaining two action sets are related to cache and storage.

5.1 Deployment Descriptors

Managing and deploying applications at the nodes requires configuration files known as deployment descriptors. These descriptors impart essential information about the application, its execution and monitoring, and its requirements in the computing environment. In ACES, the following components could be contemplated in the scope of deployment descriptors: application information, resource requirements, dependencies, deployment topology, scalability and redundancy, monitoring and management, unreliability tolerance, and statefulness.

5.1.1 Application information

Application information includes metadata concerning the application, such as its name, version, and description used to identify and distinguish the application from others in the deployed setting.

5.1.2 Resource requirements

Here, the deployment descriptor outlines the necessary application resources to function correctly at the edge, e.g., CPU, memory, storage, and network bandwidth. Resource requirements are specified for optimal performance to allocate the necessary resources accordingly.

When specifying the resource request for containers in a pod in Kubernetes, the scheduler uses this information to decide which node to place the workload on. When defining a resource limit, the kubelet—a component that runs on each node in the cluster, ensuring that all containers in the pod are running—enforces those limits so that the running workload cannot use more of that resource than the configured limit. The kubelet also reserves at least the requested amount of that system resource, specifically for that container. For example, a CPU-intensive workload should be placed on a node with sufficient CPU resources. If the workload has not specified the request for the resources, it could still be scheduled on the edge node, but it might not be able to perform its tasks as expected. Similar scenarios could happen for other resources or requirements, such as memory, local ephemeral storage, bandwidth, latency, etc.

While limits may not be essential for scheduling, they contribute towards preventing workloads from consuming more resources than planned. For example, a workload has a memory leak and is not restricted by a memory limit. In such a case, the workload could exhaust all available memory on the node, impacting other workloads on this node. Similarly, if a workload attempts to utilize the entire available bandwidth, it could cause different workloads on the node to experience network starvation.

By enforcing resource limits and requests, such situations could be prevented. Moreover, this approach also ensures that each workload has access to the necessary resources without impacting the performance of other workloads.

5.1.3 Dependencies

By specifying the services that a workload communicates with, the scheduler and re-scheduler can use this information to schedule workloads that frequently interact more closely. This improves performance by reducing latency while maintaining high availability.

Placing interacting services closer together can significantly enhance their performance in the following scenarios:

- If there are multiple services with microservices that frequently interact, it is advisable to locate the microservices of one service within the same region to improve performance.
- For workloads that are heavily dependent on a database, it is best to place them near the database to reduce latency and improve overall performance.
- Cloud workloads that receive traffic from a specific edge location should be placed within the same region to ensure efficient data transfer and low latency. Each workload could specify which other workloads communicate with (in the realm of Kubernetes, that would be a list of service names). Without modifying the Kubernetes objects, information on communication could be annotated as a serialized array in the annotations field. This approach would most likely turn out to be too tedious to annotate manually.

A different approach to modifying Kubernetes objects is to use service mesh sidecars or observe DNS traffic. This method allows for programmatically annotating workloads with their communication patterns, without directly adding the annotation to the Kubernetes objects themselves. The communication pattern information is then stored directly in the data warehouse. Although this approach only provides the re-scheduler with communication pattern information, it can still lead to increasingly optimal placements over time. The initial placement of a workload may not be optimal, but as more accurate communication patterns are determined, more efficient placement decisions can be made.

If the cluster intends to use a service mesh, Istio[6] should be considered as it already implements service-to-service communication tracking. If not, Otterize[7] network mapper can still map the service-to-service communication.

5.1.4 Deployment Topology

Deployment topology includes details about how the application will be distributed across different edge nodes and how it will interact with other deployed applications or services.

When a workload is annotated with **edge-only**, the scheduler and re-scheduler will ensure that it is placed exclusively on the edge nodes. That might be a specific node on the edge, a group of nodes in some edge location, or all nodes on the edge layer. If that is not possible (for example, when there are no available resources at the specified edge node), then the workload should not be scheduled.

The workload is solely placed on the cloud nodes by utilizing **cloud-only** annotation. This can encompass specific cloud nodes, availability zones, regions, or all available cloud nodes depending on the workload's needs.

Some workloads can run both on the edge layer and on the cloud layer, and it should be up to the scheduler or re-scheduler to determine the most optimal placement. In such cases, the workloads should be annotated with **edge-cloud** and, if needed, with a preference for some layer. In certain cases, workloads have the flexibility to run on both edge and cloud layers. In such cases, it is the responsibility of the scheduler or re-scheduler to determine the optimal placement. To facilitate this, workloads should be annotated with edge-cloud labels and, if needed, with a preference for some layer. Edge-cloud reuses edge-only and cloud-only labels.

5.1.5 Scalability and Redundancy

This specification defines how an application can scale horizontally (across multiple edge nodes) or vertically (by increasing resources within a single edge node) to handle varying workloads. Additionally, redundancy mechanisms, such as failover and load balancing, may be defined to ensure high availability and fault tolerance. Redundancy also allows the re-scheduler to perform its optimizations. There are three possible scenarios for the re-scheduler to move a workload's Pod from the origin node to a target node. In the first scenario, the target node has enough resources to create an additional pod on the target node before removing the original pod from the origin-node. In the second scenario, the nodes are fully occupied, but the workload has enough replicas to reduce the number of replicas to perform the swap temporarily. In the third scenario, the nodes are fully occupied, and the workload allows for temporary unavailability (one replica and max drawdown set to 1) in order to perform the swap. In most cases, the nodes should have enough resources to increase the number of replicas to relocate temporarily. When the nodes are full, a temporary drawdown is required to swap one or multiple pods. Workloads annotate the number of replicas in the workload's template. Allowed maximum drawdown is annotated using Kubernetes annotations at the workload level (Deployment, Stateful Set, etc.). If not specified, the default is 0.

[6] https://istio.io/
[7] https://otterize.com/

5.1.6 Monitoring and Management

Deployment descriptors can include configuration parameters related to logging, performance metrics, health checks, and other features. These settings enable the edge infrastructure to collect relevant data and monitor the application's behaviour, ensuring efficient operations.

5.1.7 Unreliability Tolerance

While operating the cluster, it may be noticed that certain nodes are more dependable than others. For instance, some edge nodes may frequently lose internet connection or experience power outages. These nodes should be labelled as such, so the workloads that are scheduled on them can explicitly acknowledge the unreliability. Only workloads that need to run on a particular edge location and those that can handle unreliability will be scheduled on these nodes.

5.1.8 Statefulness

While operating the cluster, it may be noticed that certain nodes are more dependable than others. For instance, some edge nodes may frequently lose internet connection or experience power outages. These nodes should be labelled as such, so the workloads that are scheduled on them can explicitly acknowledge the unreliability. Only workloads that need to run on a particular edge location and those that can handle unreliability will be scheduled on these nodes.

After detecting an unreliable node, it will be given a Kubernetes taint that needs to be tolerated by each workload. If the node proves to be reliable in the long run, the taint can be removed. Workloads will normally avoid tainted nodes unless they have a tolerance for the taint. However, if a workload needs to run on edge locations, it should tolerate the node's potential unreliability.

5.2 Node Affinity

Node affinity is a concept for specifying preferences or constraints regarding scheduling or placing workloads (e.g., containers, pods) onto specific nodes within a cluster. Node affinity enables to control where certain workloads are deployed based on node characteristics, labels, or other criteria. By utilizing node affinity, one can influence the placement of workloads to meet specific requirements, such as hardware capabilities, geographical location, resource availability, or custom criteria. As such, node affinity enables better resource utilization, performance optimization, and efficient allocation of workloads within a distributed computing environment.

There are two types of node affinity that both have the potential to be utilized in ACES:

- **Hard node affinity** enforces strict rules for scheduling workloads. It ensures that a workload is placed only on nodes that satisfy the specified affinity rules. If no node meets the defined criteria, the scheduling operation fails, and the workload remains unscheduled until a suitable node becomes available.
- **Soft node affinity** provides preferences rather than strict rules for scheduling workloads. It attempts to place workloads on nodes that match the defined affinity rules. However, if no suitable nodes are available, the workload can still be scheduled on other nodes that don't meet the affinity criteria.

Node affinity rules are typically defined using labels assigned to nodes and labels specified in the affinity rules. Labels are key-value pairs attached to nodes. When considering multi-cluster Kubernetes in the scope of ACES, the clusters have to be interconnected and capable of sharing a continuous stream of information with other agents across clusters. Additionally, the network infrastructure must be configured to enable workloads from one cluster to communicate seamlessly with workloads in different clusters. However, even if all the earlier requirements are met, the node affinity configurations would still not work as expected in multi-cluster Kubernetes.

To enable ACES in a multi-cluster Kubernetes environment, Karmada[8] could be utilized. Karmada introduces the concept of cluster affinity, which allows the selection of specific workloads and their placement across multiple clusters. This functionality is configured using the PropagationPolicy Custom Resource Definition (CRD). It should be researched whether it’s possible or whether Karmada and cluster schedulers could be extended to specify more granular workload placements (not only cluster), such as placement on a specific node within a particular cluster. Regarding cluster network interconnectivity, Istio Multi-Cluster can be employed to achieve network flattening. This ensures seamless communication between workloads in different clusters.

[8] https://karmada.io/

Another multi-cluster Kubernetes solution is Liqo[9], which leverages Virtual Kubelet technology to represent joined clusters as virtual nodes within the consumer cluster. With Liqo, node affinity can be used to configure workloads to either schedule on specific nodes within the consumer cluster or on the virtual nodes. In the latter case, the workloads are transferred from the virtual node to the cluster it represents. However, similar to the issue with Karmada, it seems that the granularity of node affinity in Liqo may not fully meet our requirements. Further research is necessary to determine if this limitation can be overcome or if Liqo can be extended to provide more fine-grained workload placement control. When it comes to cluster interconnectivity, it is provided by Liqo.

A potential solution to address the lack of granularity of scheduling within multi-cluster Kubernetes is to divide the scheduling process into two stages. Firstly, the initial scheduling process should evaluate the available clusters based on the information obtained from the environment. This assessment will determine the most suitable cluster at the moment. Once the most appropriate cluster has been identified, the second stage involves scheduling the workload within that cluster by selecting the optimal node within it. This level of fine-grained control allows for precise workload placement. For instance, in the case of Karmada, this approach could potentially be implemented by making modifications to the PropagationPolicy CRD to facilitate cluster selection. Simultaneously, the node affinity configuration could be utilized to define the selection of the specific node within the chosen cluster.

5.3 Helm Charts

In Kubernetes, a Helm chart represents a packaging format used to simplify the deployment as well as management of applications. It is essentially a collection of files describing the resources and configuration needed to run a particular application or service in a cluster. One can use helm charts to deploy complex applications, as they encapsulate all the necessary components and configurations in a single package.

Before deploying an application into a cluster, the process typically involves building and packaging it into a container. However, deploying the container does not provide Kubernetes with sufficient information about managing the application's workloads. Various details have to be specified, e.g., the desired number of replicas, the preferred upgrade strategy, configuration settings for the container, the target nodes for deployment, how other workloads can communicate with the workload in question, load balancing requirements, and more. To convey this information effectively, Kubernetes requires to write YAML manifests, which can become quite cumbersome, often resulting in numerous manifests for each workload. Moreover, it becomes evident that many workloads share similar Kubernetes manifests, with only minor variations. Helm addresses this drawback by means of templating to streamline the creation of these manifests. Helm simplifies Kubernetes deployments by providing a packaging format called *charts*. Charts contain meta information, configured values, templates that encapsulate all the necessary Kubernetes manifest files, and customizable parameters.

Helm improved the management of Kubernetes deployments by consolidating all the required manifests into a single Helm chart and leveraging templating to enhance reusability. When deploying a new workload, one can effortlessly reuse the same chart and customize specific configuration values.
The flexibility and reusability of Helm charts make them ideal for user sharing. These charts can be designed in such a way as to cater to various use cases, making it convenient to share them through Artifact Hub[10]—a platform that facilitates the discovery, downloading, and sharing of valuable charts.
Ordinarily, deploying a simple workload involves creating multiple Kubernetes objects like deployments, services, and config maps. Helm simplifies this process by condensing the deployment of these diverse manifests into a single, straightforward ‘helm install’ command. Similarly, removing a workload becomes effortless with the 'helm uninstall' command, while upgrading workloads is streamlined using 'helm upgrade'. Furthermore, the advantage of using Helm for workload upgrades is the seamless ability to roll back to a previous version using 'helm rollback,' ensuring enhanced reliability and risk mitigation.

Regarding ACES, it is crucial for infrastructure engineers to have a seamless deployment experience, just like deploying in any other non-ACES Kubernetes cluster. This implies that the complexities of ACES-related configurations should be abstracted or presented straightforwardly. To accomplish this goal, one can use Helm charts, which offer two promising concepts to simplify ACES workload deployments: base charts and library charts. One or both could be considered to be implemented to facilitate deploying to ACES.

[9] https://liqo.io/
[10] https://artifacthub.io/

5.4 Pod Migration

Pod migration is the process of moving a running pod from one node to another within a container orchestration system, such as Kubernetes. A specific event, such as node maintenance or hardware failure usually triggers pod migration. Utilizing this concept in ACES might contribute to maintaining high availability, optimizing the utilization of resources, and facilitating cluster management.

An overview of the pod migration process that could be utilized in ACES is the following:

1. **Event trigger**: different events, such as a node going offline or its maintenance, can trigger pod migration.
2. **Node selection**: the container orchestration system selects an appropriate target node to host the pod. The selection process could consider several factors, including resource availability, node affinity, constraints regarding the pod placement, and other scheduling policies defined in ACES.
3. **Target node preparation**: the target node has to be prepared prior to receiving the workload. Hereby, the necessary resources on the target node, such as CPU, memory, and storage, have to be available.
4. **Pod migration**: the typical scenario for migrating the pod encompasses stopping the pod on the source node and transferring its state, including network connections, storage mounts, and runtime context, to the target node. Afterward, the pod's containers are restarted on the target node.
5. **Service discovery and load balancer updates**: the container orchestration system updates relevant service discovery mechanisms, such as DNS or load balancer configurations, to ensure the optimal routing of incoming requests to the migrated pod's new location.
6. **Verification and cleanup**: verification step ensures the migrated pod is functioning correctly. This step may involve health checks, connectivity tests, or any other defined validation procedures. If the verification is successful, any remnants left on the source node are cleaned up.

5.5 Storage

Storage Volumes serve as the core abstraction in the Kubernetes storage architecture. Volumes can be permanent or non-persistent, and Kubernetes allows containers to dynamically request storage resources via a process known as volume claims.

5.5.1 Container Storage Interface

Container storage interface (CSI) is a Kubernetes addon that makes storage management easier. Prior to CSI, users had to link the data store's device driver with Kubernetes, which was a laborious process. Because CSI has an extensible plugin architecture, you can easily add plugins that support your organization's storage devices and services.

5.5.2 Volumes

Volumes are fundamental entities in Kubernetes that are used to give storage to containers. A volume can handle several forms of storage, such as network file system (NFS), local storage devices, and cloud-based storage services. To support new storage systems, you can also create your own storage plugins. Volumes can be accessed directly through pods or indirectly using persistent volumes.

5.5.3 Non-persistent Storage

Kubernetes storage is transient by default (non-persistent). Any storage defined as part of a container in a Kubernetes Pod is held in the host's temporary storage space for the duration of the pod's existence and then removed. Container storage is convenient, but it is not long-lasting.

5.5.4 Persistent Storage

Kubernetes also supports a wide range of persistent storage models, such as files, block storage, object storage, and cloud services in these and other categories. Storage can also refer to a data service, most typically a database.
Storage can be accessed directly from within a pod, although doing so violates the mobility principles of the pod and is not encouraged. Instead, pods should characterize their applications' storage requirements using Persistent Volumes and Persistent Volume Claims (PV/PVC).

5.5.5 Storage Class

A StorageClass is a Kubernetes API that allows administrators to construct new volumes as needed by configuring storage settings using dynamic configuration. The StorageClass specifies the volume plug-in, an external provider and name of the CSI driver that will allow containers to interact with the storage device.

5.5.6 Dynamic Storage Class Provisioning

Dynamic volume setup in Kubernetes allows you to construct storage volumes on demand. When a user requests a particular type of storage, the complete procedure is initiated automatically. As needed, the cluster administrator creates storage class objects. Each StorageClass denotes a volume plugin, commonly known as a provisioner. When a user creates a PVC, the provisioner builds a volume based on the storage criteria specified.

6 Conclusion

This document deals with goal representations corresponding to SLAs and SLOs for AI-/ML-/swarm-based methodology, realizing autonomy and actionability. It discusses the underlying concepts, methodologies, and objectives for managing workload placement in an edge-cloud environment with regard to specific SLAs and SLOs incorporating AI/ML/swarm-based decision-making processes. The ultimate goal is to create an autopoietic system that efficiently meets various workload demands while ensuring desired service levels.

This document has provided an in-depth overview of the goal-oriented representations and processes involved in achieving efficient workload placement within ACES. By aligning with SLAs and SLOs, ACES aims to intelligently manage the allocation of workloads across its infrastructure, ensuring the fulfilment of specific business and operational objectives related to the use cases. To this end, this document has covered some key aspects that empower ACES to deliver high-performance, secure, and reliable edge-cloud services, such as:

- Identifying relevant configuration sources and establishing SLA and SLO definitions for each use case
- Collecting and analysing configuration data and run-time metrics to inform the workload placement decisions
- Employing advanced knowledge representation for AI, ML, and swarm to enable autonomy and actionability
- Providing specific mechanisms and strategies for workload placement, cache management, and storage allocation
- Describe mechanisms for workload placement, storage and caching
- Introduce the concepts of agents and align the architecture with swarm computing concepts

By following these aspects, ACES will be able to deploy a robust and intelligent edge-cloud execution system capable of meeting diverse and demanding workload needs. This document serves as a guide for the development and implementation of ACES as a novel solution for cloud and edge computing.