Autopoietic Cognitive Edge-cloud Services

Funded by the European Union

ranche

Autopoietic Cognitive Edge-cloud Services

Deliverable 3.1
ACES Data and Knowledge Model

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

Funding of associated partners
The Swiss associated partners of the ACES project were funded by the Swiss State Secretariat for Education, Research and Innovation (SERI).

D3.1 – ACES Data and Knowledge Model	
Author(s)	Felix Cuadrado, Javier Andión, José M. Blanco, Diego Martín, Loris Cannelli, Melanie Schranz, Panagiotis Kapsalis, Fernando Ramos, João Amado, Melanija Vezocnik, Timotej Gale, Thien Duc Nguyen, Nabil Abdennadher, Konstantin Skaburskas
Editor	Felix Cuadrado
Participating partners	UL, LAKE, INESC-ID, HIRO, IDSIA, MARTEL, SixSq
Version	2.0
Status	Complete
Deliverable date	M12
Dissemination Lvl	PU - Public
Official date	31 December 2023
Actual date	20 December 2023

Executive Summary

Deliverable D3.1 – ACES Knowledge and Data Model of the ACES project presents the outcome from WP3 in identifying the information that will be required by the ACES platform to perform its management functions and capture it into a knowledge model that can be rich and usable by multiple reasoning agents. The approach described in this document aims to bridge existing gaps by offering a robust and adaptable framework to facilitate autopoietic system operations.

The ACES platform presents several unique characteristics in the edge-cloud continuum; the hardware execution infrastructure will be a collection of EMDCs (Edge Micro Data Centre), that offer disaggregated hardware resources for execution. Applications will run over a distributed Kubernetes infrastructure. The document describes the architecture for data collection within the ACES ecosystem, discussing the metrics pipeline architecture, tools for telemetry and collection, and providing specific details of the metrics and data that can be extracted from the platform.

Taking as reference these identified raw information sources the document evaluates research proposals, standards, and industry specifications to capture that data into an actionable model. In addition to that, the techniques for data cleaning, knowledge inference and creation from the base data is also presented. This includes time series analysis techniques for metrics processing, dependency and correlation analysis from multiple sources, and specific processing techniques to capture features relevant to the non-functional aspects of the system.

The central part of the deliverable describes the ACES Knowledge Model. The section discuses inherent challenges to the ACES platform for knowledge modelling, and presents a base model composed of three main elements: the supply (services execution platform), the demand (microservice and FaaS based applications), and the runtime (real time and historical status of the supply elements). To complement that information the deliverable also outlines the ACES agents that will consume this information, identifying their based types and their roles in the system.

The deliverable wraps up by discussing the main conclusions and outlines the next steps in WP3 regarding the perception of ACES agents towards the managed environment.

Disclaimer

This document contains material, which is the copyright of certain ACES contractors, and may not be reproduced or copied without permission. All ACES consortium partners have agreed to the full publication of this document if not declared "Confidential". The commercial use of any information contained in this document may require a licence from the proprietor of that information. The reproduction of this document or of parts of it requires an agreement with the proprietor of that information., according to the provisions of the Grant Agreement and the Consortium Agreement version 3 – 29 November 2022. The information, documentation and figures available in this deliverable are written by the Autopoiesis Cognitive Edge-cloud Services (ACES) project's consortium under EC grant agreement 101093126 and do not necessarily reflect the views of the European Commission. The European Commission is not liable for any use that may be made of the information contained herein.

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
15/09/2023	1.0	Table of contents	UPM
29/9/2023	1.1	Updated ToC and initial section assignments	UPM
16/10/2023	1.2	Updated structure after Ljubljana workshop discussions	UPM
20/10/2023	1.3	Input to the sections 4.2 and 4.3	LAKE
25/10/2023	1.4	Section on Framework Language and Tools goes to D4.1	UPM, LAKE
27/10/2023	1.4	Section 4.2	IDSIA
31/10/2023	1.5	Refined structure and outlines of chapters and sections, initial contributions for several chapters	UPM, LAKE, MAR
30/11/2023	1.8	Full draft version of chapters and subchapters, ready for internal review	UPM, LAKE, MAR, SIXSQ, INESC, TUD
8/12/2023	1.9	Internal Review Comments	MAR, SIXSQ
15/12/2023	2.0	Complete full version after applying reviewer comments	UPM, UL, MAR
20/12/2023	2.1	Final review	INESC

Authors

AUTHOR	PARTNER
Felix Cuadrado, Javier Andión, José M. Blanco, Diego Martín	UPM
Loris Cannelli	IDSIA
Melanie Schranz	LAKE
Panagiotis Kapsalis	MARTEL
Fernando Ramos, João Amado	INESC ID
Melanija Vezocnik, Timotej Gale	UL
Thien Duc Nguyen	TUD
Nabil Abdennadher, Konstantin Skaburskas	SixSq

Reviewers

NAME	ORGANISATION
Panagiotis Kapsalis	MAR
Vito Chianchini	MAR
Nabil Abdennadher	SixSq

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 Centers
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 ... 10

1.1 Background ... 10

1.2 Approach ... 10

1.3 Structure of the document ... 11

2. Cloud-edge Continuum Data Sources ... 12

2.1 Metrics Pipeline Architecture ... 12

2.2 Tools for telemetry and collection ... 13

2.2.1 Pull-Push Metrics Extraction Flow ... 13
2.2.2 Technologies ... 13
2.2.3 Extracted Data ... 14

2.3 Consolidated ACES metrics ... 15

2.3.1 Application-level Metrics ... 15
2.3.2 Node-level Metrics ... 17
2.3.3 Cluster-level Metrics ... 19
2.3.4 Events ... 20

3. From data to knowledge ... 22

3.1 Knowledge capture models and standards ... 22

3.1.1 Ontology languages for knowledge capture ... 22
3.1.2 Industry standards for edge continuum information ... 24
3.1.3 FIWARE NGSI-LD ... 27
3.1.4 Available datasets ... 28

3.2 Techniques for knowledge inference and creation ... 29

3.2.1 Time series analysis ... 30
3.2.2 Dependency Analysis ... 33
3.2.3 Network flow statistics processing ... 35
3.2.4 Feature extraction and modelling techniques for security and privacy ... 36

4. ACES Knowledge Model ... 38

4.1 Knowledge Model description ... 38

4.1.1 Modelling Challenges ... 38
4.1.2 Base concepts ... 40
4.1.3 Supply model ... 40
4.1.4 Demand model ... 42
4.1.5 Runtime model ... 44

4.2 ACES agent types ... 45

4.2.1 Demand Swarm Agents ... 45
4.2.2 Supply Swarm Agents ... 46
4.2.3 Orchestration of Swarm Agents with ML ... 46

5. Conclusion ... 47

References ... 48

© 2023

1. Introduction

1.1 Background

The dynamic nature of edge computing environments, characterized by low latency requirements, resource limitations, and demand volatility, presents distinct challenges. Existing models often lack the flexibility to efficiently capture these challenges and implement solutions that can reason in an effective way that improves overall system behavior. In response, the ACES data and knowledge model is designed to bridge these gaps by offering a robust and adaptable framework to facilitate autopoietic system operations.

The ACES project draws inspiration from the concept of autopoiesis [1], which refers to a system's ability to maintain and renew itself autonomously. The project's ambition lies in creating a self-managing architecture that proactively responds to external and internal variations and evolving service requirements [1]. Central to this ambition is a knowledge model that effectively captures the diverse types of data that are generated in the ACES platform, and can integrate them into a complex, interconnected model that contains all the relevant features for ACES agents and Machine Learning components to perform its function and achieve autopoietic behavior to the platform.

This deliverable reports the work primarily undertaken in WP3 - Data Acquisition, Knowledge Generation and Organization of the ACES project, in particular the effort from tasks T3.1 – From Data to Knowledge and T3.2 – Perceiving the Environment over the first year of the project. The information is further complemented with the overall definition of the ACES architecture, as reported in D2.1 - ACES Architecture definition (M12), and the ACES action library, together with the initial definition of agent approaches, as reported in D4.2 – Action language and Library (M12). These three deliverables together constitute the first view of the autopoietic approach of ACES to address the challenges behind the management of the cloud-edge continuum.

1.2 Approach

As the name of the deliverable implies, this document provides an analysis starting from the initial data sources that are involved in the ACES ecosystem, to a knowledge model that can be referred to and exploited by the different ACES components.

In order to obtain this model, the project has performed multiple activities. During the workshop taking place in Darmstadt in the first half of the year, a blueprint for the ACES system was defined. During this process a comprehensive set of potential metrics and autopoietic behavior was identified and collected. These elements were fundamental for the identification of the data that should be considered in the project. In addition to these, the members have performed thorough review of the state of the art in scientific, technical and industrial contexts to identify the most relevant information that needs to be considered, as well as the means for capturing and collecting that information.

On that basis, the WP has studied the main techniques for capturing that information into a knowledge model that would be actionable by ACES agents. This deliverable reports the results of this analysis, and the resulting core abstractions of a knowledge model that will be combined with the different types of ACES agents to achieve the autopoietic characteristics of the system.

1.3 Structure of the document

The structure of the document follows the logical progression presented in the approach. We start from the theoretical underpinnings to its practical application. It comprises the following sections:

Chapter 2 covers the **Data Sources** that can be collected from the ACES architecture. These are expressed in the form of metrics of the runtime elements that form part of the overall ACES architecture, as well as events that should be recorded and processed for further analysis. The information is complemented with details on the architecture of the specific monitoring and data collection components of the architecture. As information captured by telemetry systems can be in a raw and hard to use state, data processing, cleaning and analysis techniques are also presented. Moreover, an examination of the requisite architectural components and mechanisms for data acquisition and the aggregation of metrics relevant to the ACES project are presented.

Chapter 3 explores the challenges from this transition **from Data to Knowledge**. The subsections report a discussion on the models, standards, and methodologies employed for knowledge derivation and synthesis within edge computing frameworks.

Chapter 4 presents the **ACES Knowledge Model**. This is the central outcome of this deliverable. The chapter details the fundamental aspects of the knowledge model, encompassing the supply and demand models and the operational runtime model crucial for the autopoietic operations of the ACES platform. To complete this characterization, the section provides a definition of the types of agents that will take advantage of this model and reason within the ACES platform.

The deliverable is completed with the main conclusions of this work, and an outline of next steps within the work related to the perception of ACES environments.

2. Cloud-edge Continuum Data Sources

ACES components will gather telemetry data from EMDCs. To achieve this objective, we will focus on the activities outlined in WP3 – "Data Acquisition, Knowledge Generation, and Organization." One of the key deliverables within this work package is the ACES Metrics pipeline. This pipeline is specifically designed to capture data from Kubernetes clusters, nodes, and applications hosted in EMDCs. Therefore, our approach to build the pipeline will consider technologies related to Kubernetes environments. Before delving into the technical components that will be utilized in the pipeline, it's essential to highlight the data sources within EMDCs. We categorize these data sources into two broad categories:

- **Metrics in Kubernetes:** System component metrics originated from Kubernetes components like:
    - Kube-controller-manager
    - Kube-proxy
    - Kube-apiserver
    - Kube-scheduler
    - Kubelet
- **Application Metrics:** Metrics scrapped from applications running in Kubernetes clusters in target micro EMDCs.
- **Network Metrics:** These metrics are collected directly from the host network stack and/or from the EMDC network switches.

The section is structured as follows. First, we present the Metrics pipeline architecture. Then, we analyse the tools for telemetry and collection. Finally, we list the target ACES metrics.

2.1 Metrics Pipeline Architecture

The figure below illustrates the Metrics pipeline architecture followed in ACES:

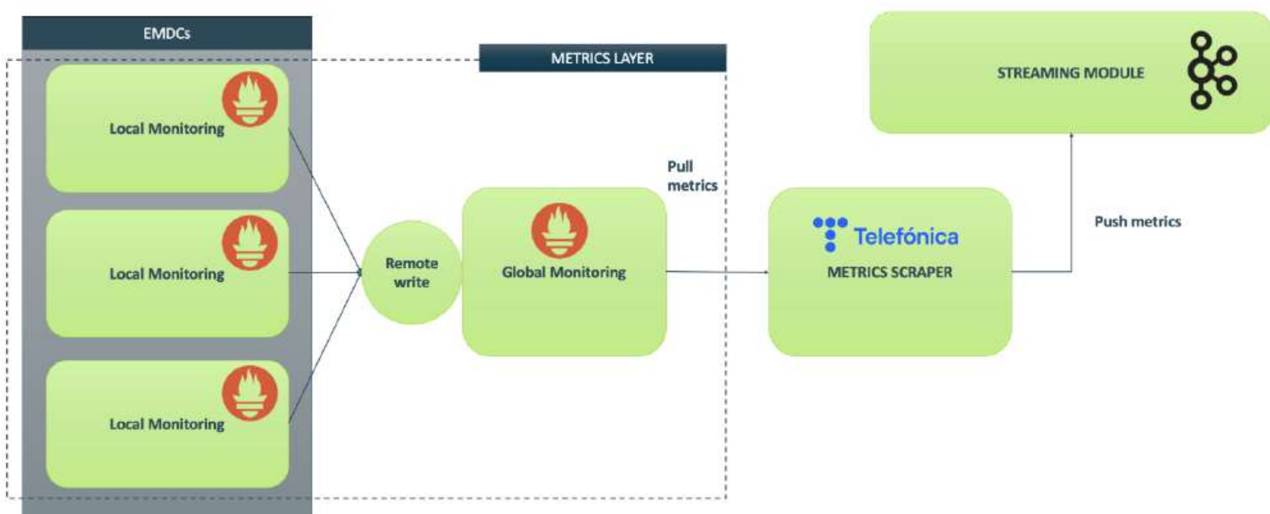

Figure 1 Metrics Pipeline Architecture

In the context of the Metrics Pipeline Architecture, EMDCs' metrics are collected through open-source monitoring systems within each EMDC, such as a Kubernetes cluster. These may also include metrics collected outside the host, namely those from in-switch metric collectors. Additionally, an abstraction

layer exists as the "global" component, which consolidates metrics from the various "local" data monitoring systems. The primary role of this "global" monitoring component is to gather and transmit the extracted metrics to a metrics extraction job. This job is responsible for receiving the metrics and forwarding them to a streaming component for further processing.

2.2 Tools for telemetry and collection

2.2.1 Pull-Push Metrics Extraction Flow

The ACES Metrics Pipeline follows a "pull" – "push" architecture, in particular Local Monitoring modules are placed in every EMDC to extract telemetry data. Then local monitoring performs remote write operation to expose the local metrics to a Global monitoring instance which is responsible to assemble the metrics from EMDCs. Then a component pulls the metrics from Global monitoring and produces them to a Streaming Module (Apache Kafka). This flow is depicted in the following figure.

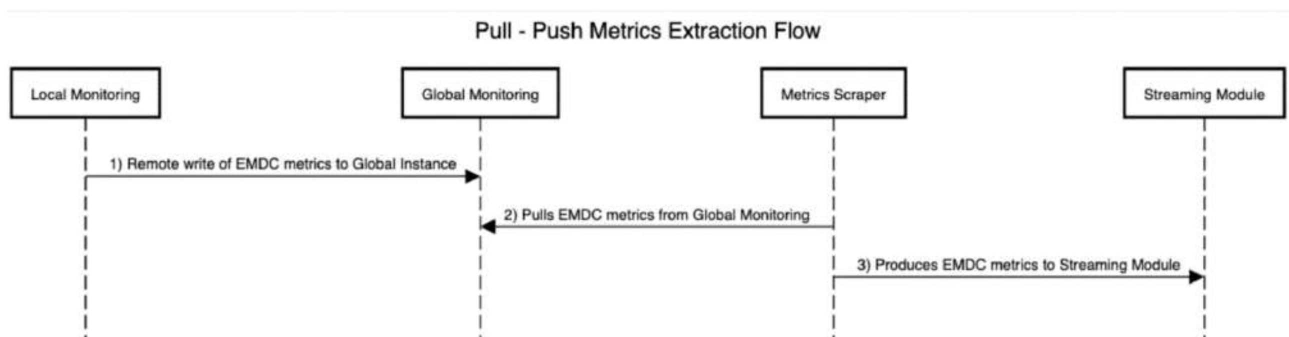

Figure 2 Metrics Extraction flow

2.2.2 Technologies

The tools that are used for metrics collection are the following:

- **Local Monitoring (Monitoring System in each EMDC):** Prometheus[1] and its building blocks like alert-manager, kube-state-metrics, node-exporter, Prometheus-pushgateway, Prometheus server, and network metrics (e.g., in-switch flow metrics).
- **Global Monitoring (Prometheus Hierarchical Mode[2]):** Hierarchical federation allows Prometheus to scale to environments with tens of data centers and millions of nodes. In this use case, the federation topology resembles a tree, with higher-level Prometheus servers collecting aggregated time series data from a larger number of subordinated servers. To enable federation mode it is needed to include the following lines of code to Prometheus configuration file (Prometheus.yml)

Table 1 Activate Federation Mode in Prometheus

```
scrape_configs:
  - job_name: 'federate'
    scrape_interval: 15s
```

[1] https://prometheus.io/
[2] https://prometheus.io/docs/prometheus/latest/federation/

```
          honor_labels: true
          metrics_path: '/federate'

          params:
            'match[]':
              - '{job="prometheus"}'
              - '{__name__=~"job:.*"}'

          static_configs:
            - targets:
              - 'source-prometheus-1:9090'
              - 'source-prometheus-2:9090'
        - 'source-prometheus-3:9090'
```

- **Metrics Scraper:** Metrics from the micro datacenters are produced to a Streaming module using Prometheus Kafka Adapter[3], which is a service that receives Prometheus metrics through remote write functionality into JSON and sends them to Apache Kafka (Streaming module)

Table 2 Remote write to Prometheus Kafka Adapter service

```
      remote_write:
        - url: "http://prometheus-kafka-
      adapter:8080/receive"
```

- **Streaming Module**: Confluent Kafka[4] used and its building blocks:
  - **Confluent Kafka Broker**: The Kafka broker is the most important component as it maintains the topics and the different partitions.
  - **Apache Zookeeper**: It is used to manage and coordinate the Kafka brokers in the cluster.
  - **Confluent Control Center:** Is a web-based tool for managing and monitoring Apache Kafka in Confluent Platform. It provides a user interface that enables overview of cluster health, topic and messages observation, Schema Registry configuration and the development of ksqlDB queries.

2.2.3 Extracted Data

The extracted data are pipelined to a Streaming Module through the Metrics Scraper (Prometheus Kafka Adapter) which is a service that receives Prometheus metrics through remote write operation configured in Prometheus configuration and each metric is sent to Kafka broker periodically, (e.g., every 30 seconds). The schema that is used from Metrics Scraper is the following:

Table 3: Metrics Abstract Schema

```
      {
          "namespace": "io.prometheus",
          "type": "record",
          "name": "Metric",
          "doc:" : "A basic schema for representing Prometheus metrics",
          "fields": [
              {"name": "timestamp", "type": "string"},
```

[3] https://github.com/Telefonica/prometheus-kafka-adapter
[4] https://www.confluent.io/

```
            {"name": "value", "type": "string"},
            {"name": "name", "type": "string"},
            {"name": "labels", "type": { "type": "map", "values": "string"} }
        ]
    }
```

Table 4: Example Record

```
    {
      "labels": {
        "__name__": "go_memstats_mcache_inuse_bytes",
        "instance": "prometheus-prometheus-pushgateway.default.svc:9091",
        "job": "prometheus-pushgateway"
      },
      "name": "go_memstats_mcache_inuse_bytes",
      "timestamp": "2023-11-27T13:36:40Z",
      "value": "4800"
    }
```

2.3 Consolidated ACES metrics

Metrics collection enables the ACES platform to acquire real-time information about its current state. These data and past metrics information represent the fundamental basis for making informed workload placement decisions and adapting to environmental and demand changes.

Metrics can be categorized into three levels within the ACES platform hierarchy. At the first (top) level, there are one or multiple clusters. Each cluster contains multiple nodes at the second level. Nodes also contain multiple workloads at the third level. Considering these levels, we propose the following categories of metrics: application-level, node-level, and cluster-level. At each level, we specify the raw metrics that should be collected where feasible. Additionally, we offer some ideas of metric aggregations and transformations that may be more beneficial than the raw metrics. It is crucial to elucidate the rationale behind opting for raw metrics rather than aggregations. By collecting metrics in their purest form, we can give the platform users the utmost flexibility to aggregate and transform the data according to their specific business requirements, for example, when creating SLIs. Moreover, the platform might initially employ one metric aggregation and later switch to a different, more effective aggregation. If the platform were to collect only the aggregated form of the metrics, such a transition would not be possible. To ensure smooth interoperability and adhere to a standardized approach, we recommend collecting the specified metrics (and traces) in alignment with the OpenTelemetry[5] standard.

For each metric level, we have identified the following raw metrics and some example aggregations and transformations.

2.3.1 Application-level Metrics

Application-level metrics represent the platform's most focused metrics, offering a detailed perspective on the pod's state over time. This information is vital for confirming that the pod operates as anticipated. In cases where deviations occur, these metrics assist in identifying necessary actions to correct the situation and ensure optimal performance. Table 5 includes a list of application-level metrics.

[5] OpenTelemetry, https://opentelemetry.io/docs/

Table 5: List of application-level metrics.

METRIC	METRIC NAME	DESCRIPTION
Pod CPU usage	*kube_pod_overhead_cpu_cores*	*Current pod CPU usage measured in millicores. The rescheduler could take the historical variability of this metric into account to schedule the pod on a node that has the resources needed to handle the CPU usage spikes or usage that is commonly higher than the requested amount. Additionally, this metric could be used in scaling decisions.* *A possible transformation of this metric is CPU percentage usage, where 100% could either be the CPU request of the CPU limit of the Pod.*
Pod memory usage	*kube_pod_overhead_memory_bytes*	*Memory usage of the pod measured in bytes. This metric could be used in scaling decisions.* *Similarly, as the CPU usage, a percentage-based transformation can be created.*
Pod GPU usage	*kube_pod_overhead_gpu_cores*	*Current pod's GPU usage measured in millicores.*
Pod ephemeral storage usage	*ephemeral_storage_pod_usage_bytes*	*The current pod's ephemeral storage usage measured in bytes.*
Pod total readiness and health checks - with failure bool label	*kube_pod_status_ready* *kube_pod_status_phase* *kube_pod_status_reason*	*Each probe's result is reported as a metric. This metric could be used to construct an SLI that is part of an SLO addressing availability and fault tolerance. The scheduler should take such metrics into account and try to satisfy the SLOs.*
Request duration	*request_duration_milliseconds*	*The duration of the request. This metric could be used to construct SLIs and SLOs. The platform should try to achieve the desired SLO by, for example, scaling and moving some pods closer so that the latencies between them are reduced. To determine which part of the call chain to optimize, the scheduler would require trace information besides the metrics. Some possible aggregations of this metric are request duration per second, percentage of total request durations that are larger than 1 second, etc.*
Total requests and no. of concurrent requests	*requests_total*	*This metric could be used to construct SLIs and SLOs. The platform should try to achieve the desired SLO by, for example, scaling and moving some pods closer so that the latencies between them are reduced. To determine which part of the call chain to optimize, the scheduler*

		would require trace information besides the metrics. Based on the total requests metric, a throughput metric can be constructed, for example, the number of requests per second.
Garbage collection metrics	*memstats_gc_sys_bytes memstats_last_gc_time_seconds memstats_next_gc_bytes*	*If no memory limit is set, the platform could detect memory leaks based on the garbage collection results (along with heap usage metrics).*

It is important to note that for the ACES platform to fully understand the complexities and interrelations of pods, relying solely on metrics is insufficient. In order to gain insights into communication patterns, microservice call bottlenecks, and more, traces play a crucial role in observability. Just like metrics, traces should adhere to the OpenTelemetry specification for comprehensive data collection.

2.3.2 Node-level Metrics

Node-level metrics provide information on the current resource usage of the node, aiding in the calculation of available resources per node, which plays an essential role in workload placement decisions. Additionally, node-level metrics offer insights into the node's connectivity with other nodes, resource costs, and node reliability. Table 6 lists node-level metrics.

Table 6: List of node-level metrics.

METRIC	*METRIC NAME*	*DESCRIPTION*
Node CPU usage	*node_cpu_usage_seconds_total*	*Node’s CPU usage measured in millicores. Based on the node’s total CPU core descriptor, the available CPU can be calculated. The scheduler should take this metric into account in order to place only pods that request less than or equal available CPU amount on such nodes. Furthermore, the scheduler could, with the CPU usage metric among other metrics, ensure that the load is evenly spread across nonelastic (edge or reserved) nodes. In this case, the transformation to the percentage-based CPU usage could be used.*
Node memory usage	*node_memory_working_set_bytes*	*Node’s memory usage measured in bytes. Based on the node’s total memory descriptor, available memory can be calculated. The scheduler should take this metric into account to place only pods that request less than or equal available memory on such nodes. Furthermore, the scheduler could, with the memory usage metric, among other metrics, ensure that the load is evenly spread across nonelastic (edge or reserved) nodes.*

Node GPU usage	*node_gpu_usage*	*Current node's GPU usage. How this can be measured, highly depends on the GPU's manufacturer. The scheduler should take this metric into account in order to provide a sufficient amount of GPU power to each pod. Furthermore, Pods should be scheduled in a way that they can take advantage of all the GPUs available in the cluster.*
Node ephemeral storage usage	*ephemeral_storage_node_usage_bytes*	*The current node's ephemeral storage usage measured in bytes. Based on the node's total ephemeral storage descriptor, available ephemeral storage can be calculated. The scheduler must ensure that each pod has the requested amount of ephemeral storage.*
Node swap usage	*node_swap_usage_bytes*	*Node's swap usage measured in bytes. The platform should try to either avoid or minimize swap usage.*
Node to nodes latency	*node_latency_milliseconds*	*Each node could periodically perform pings to other nodes in the cluster to determine latencies, measured in milliseconds, between them. Scheduler can use this metric to schedule Pods that frequently communicate or require low latencies for their communication, closer than the rest of the pods. Furthermore, when performing periodic pings, some pings might fail. Based on this information, a voting algorithm can be constructed that outputs the node's reliability. This information can be used by the scheduler to place pods that can easily handle disruptions on nodes that are not as reliable and place pods that are either critical or cannot gracefully handle disruptions on nodes that are reliable. Edge nodes are apparent candidates for unreliability, along with spot nodes in the cloud.*
Node to nodes bandwidth usage	*node_bandwidth_bps*	*Each node could measure how much egress bandwidth, measured in bps, is being used. This information can be used to calculate the available bandwidth, but only if the following can be performed without causing too many disruptions to the running pods. Each node can measure the maximum egress bandwidth (or probably available bandwidth) to the rest of the nodes, and it can ask other nodes to gather information about the maximum (available) ingress bandwidth. The scheduler can use this information to reduce the bandwidth usage between*

		nodes, but at the same time, ensure required fault tolerance levels and provide each pod the requested bandwidth.
Node cost	node_cost_aps	Cloud provider's node costs vary based on many different factors. Each node in the cloud should output how much it costs per second to run it (measured in amount per second). This should include IO, storage, compute, and other costs which should be reduced to a single cost metric. Edge nodes would have to measure their power usage, networking costs, amortized hardware costs, maintenance costs, etc., and reduce this to a single cost metric. The scheduler should use this metric to reduce the operating costs, but at the same time, take into account all the required requirements (fault tolerance, SLOs, etc.). For example, this could mean reducing the load in the cloud in order to reduce the required number of nodes, which would result in lower cost or deprioritizing edge nodes where the electricity cost is high.
System call statistics	node_syscall_{type}_total	Number of system calls triggered per a time interval, frequently called of a system call, frequently called of a type of system calls (e.g., open, read, write, exec, exist, kill)

2.3.3 Cluster-level Metrics

As a cluster consists of many nodes, cluster-level metrics predominantly serve as aggregations of node-level metrics. The utility of these metrics in workload placement and scaling decisions may need to be revised, given that aggregations often simplify the underlying node-level specifics. However, from the standpoint of platform administrators, these metrics could be monitored by means of monitoring tools, such as the Grafana dashboard[6], or directly in network switches. Table 7 shows a list of cluster-level metrics.

Table 7: List of cluster-level metrics.

METRIC	METRIC NAME	DESCRIPTION
Network flow types	network_flow_type	Network metrics computed for different flow types. Examples include [MAC src, IP src], [IP src], [IP src, IP dst], and [5-tuple]. Different flow types enable different aggregation levels, from coarser to finer-grained monitoring modes.
Network flow counters	network_{type}_total	Per-flow counters for the different flow types. Examples include number of

6 https://grafana.com/grafana/dashboards/6417-kubernetes-cluster-prometheus/

		packets, number of bytes, squared number of bytes. Maintaining these stateful counters in network switches offers global visibility for all communications at the EMDC level.
Network flow statistics	*network_{type}_statistics*	*Unidirectional statistics, tracking outbound traffic, including weight, mean, std. deviation, time interval. Bidirectional statistics, considering both inbound and outbound traffic: magnitude, radius, approx. covariance, and correlation coefficient. Computing different statistics enable support for different monitoring applications (e.g., anomaly detection, malicious traffic detection, traffic engineering, etc.)*
Cluster total resource usage	*cluster_cpu_usage_seconds_total* *cluster_memory_working_set_bytes* *cluster_gpu_usage* *cluster_bandwidth_bps*	*An aggregated metric that is calculated as a sum of node-level resource usages. For example, total CPU usage, total memory usage, total GPU usage, total bandwidth usage, etc. This metric could be useful, for example, if the goal of the platform is to reduce the bandwidth usage within the cluster. Or, to provide an overview of the platform's resource usage which to provide an overview of the platform's resource usage, which can be perhaps used to ensure that the load is evenly distributed across many clusters. Both the aggregated metric and the transformed percentage-based metric can be considered.*
Cluster total cost	*cluster_cost_aps*	*An aggregated metric calculated as a sum of node-level memory usages. Provides an overview and can be used to track the workload placement optimization effectiveness.*
Unhealthy pods in the cluster	*pod_collector_zone_health* *pod_collector_unhealthy_pods_in_zone*	*Number of unhealthy pods in the cluster.*
Unhealthy nodes in the cluster	*node_collector_zone_health* *node_collector_unhealthy_nodes_in_zone*	*Number of unhealthy nodes in the cluster.*

2.3.4 Events

Certain events could be pivotal in optimizing the operation of the ACES platform. These events could encompass a spectrum of interactions ranging from service-initiated actions to security-related events. Additionally, monitoring certain events in ACES could lay the groundwork for analytics, enabling to gain insights into specific behaviors and refine the design and features. Table 8 lists events that have the potential to be monitored in ACES.

Table 8 List of events.

EVENT	DESCRIPTION
Failure or error	*Failure or error events in the system are deemed most crucial as they have critical implications for availability and stability. Examples of such events are image retrieval errors or incorrect configuration errors.*
Workload eviction	*During system operation, the orchestrator may terminate specific pods due to several reasons, such as insufficient resources. Monitoring and understanding these events are necessary to optimize system configuration and resource misallocations.*
Workload scheduling	*Events related to scheduling (e.g., scheduling failure) provide insight into resource provisioning and enable rectifying the scheduling configuration or resource scaling.*
Storage	*Workloads and applications normally rely on external/volume storage to store data and support runtime. Storage related events indicate potential issues with mounting, attaching, capacity or other general storage failures.*
Network	*Workloads, storage, pods, nodes, and clusters are generally interconnected via a network. Network events, such as link down event or bandwidth limit exceeded event, enable dynamic reconfiguration, rescheduling, or more optimal traffic rerouting.*
Workload/Application-specific	*Workload and application events provide insight into instrumented runtimes, which enable observability of state and potential issues, e.g., application bugs or performance.*
Node-specific	*Node events provide insight into behaviour of nodes in the system. Such events may relate to specific node issues (e.g., node unhealthy, node not ready, port conflict) or aid in maintaining observability for other components (e.g., node reboot event).*
Cluster-specific	*Cluster events provide insight into behaviour of clusters in the system. Such events may relate to specific cluster issues (e.g., cluster control plane unhealthy) or aid in maintaining observability for other components (e.g., cluster uptime).*

3. From data to knowledge

After presenting the set of considered data sources that are being considered at ACES, in this section we present the alternatives for capturing that knowledge. We first explore existing standards for capturing that information in a cohesive, semantically expressive format that can be adjusted to the requirements of the project. Following that, we present potential data processing techniques that can take as input the raw metrics described in the previous section and transform them into the knowledge required by agents to perform their reasoning.

3.1 Knowledge capture models and standards

In this section we analyze the existing alternatives for capturing the knowledge relevant for the ACES agents. We explore both standards from the academic literature in the field of ontologies as well as the main approaches followed by industry and research projects, in order to select the ideal abstractions to capture all the relevant characteristics.

3.1.1Ontology languages for knowledge capture

Ontologies are defined as a mean to formally model the relations and entities of a determined structure or domain. That is, an ontology is to depict the internal structures that can include both entities and relations. To develop and deploy them, several specialized languages have been used for a long time [3]. At first, the languages used were nothing but variations of web languages, or even web languages themselves, highlighting the connection of ontologies with the semantic web. Amongst the ones that are web languages by themselves, it is possible to find XML (eXtended Markup Language), RDF (Resource Description Framework), or RDF Schema. The main advantage of using these is that they have no learning curve for anyone familiar with them. The downside, as obvious as it is, is that they lack the specialization and precision that others might have. It is also worth noting that these languages are built one upon another: RDF Schema upon RDF, and RDF upon XML. This leads to each one being more expressive than the previous ones, having a more extensive syntax means that they cover a larger ground.

At the other end of the spectrum, we find the more traditional ontology specification languages. Among these we find Ontolingua, OKBC (Open Knowledge Base Connectivity), OCML (Operational Conceptual Modeling Language), FLogic (Frame Logic), and LOOM [3]. Since these languages were developed with the creation of ontologies in mind, they have many resources that are ontology-oriented, including rules for reasoning, the ability of performing non-monotonic inferences, as well as ensuring the desired level of expressiveness for the
underlying logic. Despite all this, some of the languages lack the pragmatism that would be desirable for the quick development of ontologies and have been more focused on being a solid foundation to ensure the correct performance of the ontologies that use them.

As a combination of all these previous languages it is possible to find the web-based ontology specification languages. These languages, while maintaining the main core of web-based languages, were introduced to have a much higher appeal to the development of ontologies that the traditional ontology languages have. Among these is possible to find XOL (XML-based Ontology exchange Language), SHOE (Simple HTML Ontology Extension), and OIL (Ontology Interchange Language). These languages aimed at solving the issues that the previous categories had, i.e., web languages lack of specialization, and traditional ontology languages lack of quick applicability. Usually, these languages were developed as a subset of previous languages, as in the case of XOL, which takes a subset of OKBC and includes some of the characteristics and syntax of XML, or OIL based on subsets

of RDF, and OKBC. This leads to them having more value from an ontological point of view yet being more applicable than the pure web languages above.

Nevertheless, despite all of the above, the ontology languages that have become a key element in the development of the modelling that represent ontologies are RDF Turtle[7] and OWL (Web Ontology Language)[8], aside from all their offspring. The first one, RDF Turtle, stems from the RDF web language incorporating a new syntax called Turtle that allows the RDF generated graph to be processed in compact and plain text which includes abbreviations for the most common data types and patterns that are usually used. It also provides compatibility with other languages such as SPARQL following the W3C recommendations. On the other hand, OWL, that stems into OWL Lite, OWL DL, and OWL Full (among others), is characterized for using a formal syntax akin to that of RDF but has found its way into healthcare research. The language has gained an unusual level of traction and has helped in the development of its second version, OWL 2, that included several improvements proposed by the community. Among these advantages we find the compatibility with ontology standards such as Protégé[9], or established semantic reasoners such as Pellet[10], HermiT[11], or some minor ones such as RacerPro[12].

Such languages created to help with the development of ontologies are able to model “almost anything”, one of the main reason that they are actually such a powerful and useful tool. But that also has its own drawbacks, as the modelling of the specifications becomes a task of utmost importance that can lead to certain critical failures if it is not addressed correctly. Because all of this, the modelling of the specifications has become a hot topic for ontologies, which has seen some interesting results in that respect. Some include the development of specification languages that ensure the creation of ontologies without any kind of inconsistencies, errors, or ambiguity. An example of this can be found in ReqDL[6], a language whose syntax was developed so engineers can express the system requirements in a simple and easily understandable way, yet far away from the issues derived from using natural language. In particular, the language is developed with the idea of having the different requirements attached to different levels. That way it can distinguish, among others, between the stakeholders requirements, the system requirements, and the component requirements, while keeping the same syntax for all of them. This leads to a clear and streamlined syntax that can capture the requirements without cluttering them and being able to formalize them with ease.

This work of establishing the specifications is not only supported by specific ontology languages like ReqDL above, but also uses ontologies themselves to cover the structure that the specifications should have, dividing them into the different abstraction levels, and ensuring that the relations created are the right ones. For that matter, the ontology usually provides support for a specific domain and focus, ensuring that no question outside of the domain at hand goes into the requirements specification. Furthermore, it helps with the formulation of competence questions that constitute the validation process of any requirements specifications. After all of this, often an ontology is created once the process has been followed. That means that the ontology created captures all the requirements, and that those requirements have been modelled with all the benefits that correspond to the usage of ontologies.

Similarly, the creation of ontologies has its own model for specifications, that while is not of use in the capture of the requirements of any other domain, allows for the creation of ontologies that will

[7] https://www.w3.org/TR/turtle/
[8] https://www.w3.org/TR/owl-features/
[9] https://protege.stanford.edu/
[10] https://github.com/stardog-union/pellet
[11] http://www.hermit-reasoner.com/
[12] https://franz.com/agraph/racer

validate the specifications given. In that sense, it could be argued that this is a meta-modelling of the specifications. This ontology modelling is supported by the W3C and, as such, it constitutes an official modelling tool for the specifications of ontologies. Going into details, this provided model introduces a series of modules that support the lexicon on which ontologies are defined, and a series of tools to ensure the well-being of the ontology just created. The modules that constitute the lexicon include tools to deal with the syntax and semantics, the decomposition, the variations and translations, and the linguistic metadata. The other tools include the ability of using external ontologies to the one being created, a method to use linguistic resources such as lexical nets, and a way to create a relation between the lexicon chosen for the ontology and some basic languages such as the Simple Knowledge Organization System, the Lexical Markup Model, and the Open Annotation Model [4].

Despite ontologies being a great tool, they are far from perfect, as they usually require high expertise or a deep knowledge of certain formalisms and frameworks to make them work correctly. With that in mind it is possible to look for alternatives that help to capture knowledge on a more streamlined way, and with less hassle involved. For that matter, one would need to address what part of the work that ontologies do is going to be substituted. Because the way ontologies work is not only by capturing the knowledge, but also ensuring its consistency and explicitating its structure. That is, one might want to preserve the formal aspect that ontologies bring to the knowledge that they represent, or maybe the interest will reside on making the relations that they represent widely, and quickly, available. In the case that one might want to preserve the formal aspects, it is possible to be looking at technologies that focus on sharing the inferences of the ontologies, that is, helping to show what conclusions come naturally from the knowledge captured by the ontology. On sharing the semantic knowledge hold by the ontologies, that is, to ensure the inferences made from an ontological structure can be reused for other applications without uncertainty. On sharing group knowledge, that is, to ensure that the right knowledge generated by ontologies taking place on a large scale is selected, rather than mess around with the many different sources available. Nevertheless, this kind of technologies is yet not widely available or has not been developed far from its formalisms. The most extended one are the ones named as knowledge brokers[7]. Knowledge brokers are usually seen as an intermediary between the different sources of information and can help to provide the right data as needed, where and when needed. They work by transferring and exchanging knowledge from where it is abundant to where it is lacking. This means that knowledge brokers are not the originators of knowledge, but rather a technology to be used to capture knowledge that already exists and, by its usefulness, needs to be transferred.

The most common theme around knowledge capture system is that they are based around the application of ontologies[8]. This means that while many different systems based on the previous approaches have been developed, they are always, at least, partially inspired by the ontological approach. Besides using an ontology-inspired structure directly, other approaches reuse the description logics developed for ontologies, thus capturing the formal intention of ontologies. All in all, the knowledge capture alternatives to ontologies are just a differentiated application of them[9].

3.1.2 Industry standards for edge continuum information

This section describes a set of Kubernetes based edge management tools that support data management platforms. In our context, Data management means a range of functionalities that extract knowledge from telemetric data collected from EMDCs and data describing the specification of the end-user applications. The data management platform to adopt in ACES will be used to

transform data into knowledge (semantic enrichment). In this subsection we present a selection of edge management tools: Rancher[13], Zededa[14], Spectro cloud[15], OCM[16] and Nuvla[17].

3.1.2.1 Rancher

Rancher is an open-source container management platform to deploy and manage Kubernetes clusters. Rancher supports multiple Kubernetes distributions and provides full support for the two lightweight Kubernetes platforms: K3s and RKE2[18].

The incorporation of Prometheus with Rancher enables Rancher to perform monitoring and alerting functions on clusters. While Rancher lacks a native feature specifically designed for in-depth data operations, the tool provides the flexibility to develop Rancher extensions, allowing users to customize and enhance its capabilities according to their specific needs.

3.1.2.2 Zedada

Zededa's platform is designed to enable the deployment of virtualized applications on edge devices. The Zededa Edge solution comprises two key architectural components: EVE-OS and ZEDCloud.

EVE-OS boasts features such as broad hardware support, accommodating CPUs, GPUs, FPGAs, and popular selections from AMD, Intel, NVIDIA, Xilinx, and ARM. Currently, EVE-OS is validated to operate on 80 distinct hardware models and includes a fundamental open-source controller within Project EVE.

The ZEDCloud, functions as the centralized management and control plane and employs an open orchestration API to connect with EVE-OS deployed on distributed edge hardware. As for the data management and as detailed in ZEDEDA web site, a user application is represented as a metadata manifest. This manifest describes the various software pieces and how they are run on any given ZEDEDA Edge Node. The Metadata manifest is typically defined by the application developer or software provider. It describes the purpose of the application, the intended usage, and the required resources and services to run it.

3.1.2.3 SpectroCloud / Palette

Spectro Cloud offers a cloud-native management platform named Palette designed to streamline the deployment, administration, and scaling of Kubernetes clusters in various environments, encompassing on-premises and multi-cloud configurations. The platform is geared towards providing a cohesive and standardized management experience for Kubernetes clusters, irrespective of their deployment locations.

Palette features a graphical user interface (GUI) that serves as an interface for research-oriented data management, encompassing meticulous metadata management. The GUI provides users with a visual platform for efficient and intuitive interactions with data, streamlining various tasks related to

[13] https://www.rancher.com/
[14] https://help.zededa.com/hc/en-us
[15] https://www.spectrocloud.com/
[16] https://open-cluster-management.io/
[17] https://nuvla.io/ui/
[18] https://docs.rke2.io/

research data management. Moreover, Palette is equipped with an Application Programming Interface (API) designed to expose data programmatically.

3.1.2.4 Kermada

Karmada, also known as Kubernetes Armada, serves as a Kubernetes management system that allows the deployment of cloud-native applications across various Kubernetes clusters and cloud environments without requiring modifications to the applications themselves. Leveraging Kubernetes-native APIs and offering sophisticated scheduling capabilities, Karmada facilitates a genuinely open and multi-cloud Kubernetes experience. Karmada is location agnostic and supports clusters in the public cloud, on-prem, or edge.

Karmada lacks built-in data management capabilities. Prometheus can be employed to interact with the API and gather valuable insights into the system's performance and health. While Karmada may not have native data management features, its API compatibility and integration with tools like Prometheus provide a pathway for users to extract and analyze relevant data for monitoring and optimization purposes.

3.1.2.5 Nuvla

Nuvla is an edge and a container management platform built upon open-source software and open standards. The Nuvla platform allows you to configure any number of Container-as-a-Service (CaaS) (e.g. Docker Swarm, Kubernetes) endpoints. This means you can mix and match public clouds, private clouds and infrastructure, as well as edge devices (running NuvlaEdge software, see details below).

The Nuvla platform exposes a powerful REST API. This API allows developers to integrate Nuvla into third-party systems, script it and even use it as Infrastructure as code (IaC). This enables a simple and effective edge-to-multi-cloud solution. The platform is application centric, hardware agnostic, cloud neutral and container native. This allows end users to manage any containerized application across a fleet of edge devices and container-orchestration engines.

The NuvlaEdge software aims to provide a platform for managing and orchestrating edge computing resources. It helps organizations deploy and manage applications at the edge of the network efficiently. This can be especially valuable in scenarios such as Industrial Internet of Things (IIoT), smart cities, and other use cases where distributed computing and rapid data processing are critical.

Once installed, NuvlaEdge is a turn-key solution. From factory settings, you plug it in, power it up and you are good to go. The automated and secured registration process ensures that each edge device is yours and uniquely configured and initialized. This can even include an on-demand remote secured VPN access gives you access to your devices and applications as you need it.

In addition, Nuvla supports a data management platform that leverages the positive attributes of S3-based services and introduces a comprehensive global management system for metadata. The goal is to enhance the efficiency of search functionalities across different service providers. In terms of implementation, the model consists of three core Nuvla resources:

- data-object: This resource acts as a proxy for data stored in an S3 bucket/object from a specific provider. It manages the lifecycle of S3 objects, simplifying data upload and download processes.
- data-record: This resource allows users to add additional, user-specified metadata for an object. Enabling the attachment of rich, domain-specific metadata to objects enhances the precision of searching for relevant data.

- data-set: This resource defines dynamic collections of data-object and/or data-record resources through filters. Administrators, managers, or users can define these collections, providing a flexible and customizable approach to data organization.

Collectively, these resources establish a versatile data management framework applicable to a broad range of use cases. The typical workflow involves creating a data-object (implicitly creating the S3 object), optionally adding metadata using a data-record object, and finally, finding and using the relevant data-object resources included in a data set. Nuvla facilitates the "using" element by binding data types to user applications capable of processing the data, offering seamless integration between data management and application utilization.

3.1.3 FIWARE NGSI-LD

NGSI-LD[19] is a standard designed to foster better interoperability and information exchange across IoT platforms. The language has been standardized at the European Telecommunications Standards Institute (ETSI). NGSI-LD builds upon the legacy of the NGSI-9/10 interface, which was initially proposed in the FIWARE[20] platform, a suite of public, royalty-free, and open-source software components to create smart applications.

The "LD" in NGSI-LD stands for Linked Data, which is a method of publishing structured data so that it can be interlinked and become more useful through semantic queries. NGSI-LD facilitates the representation, exchange, and querying of context information across different systems. It leverages the power of linked data and semantic web technologies, such as Resource Description Framework (RDF) and Web Ontology Language (OWL), to enhance the capabilities of context-aware systems. Moreover, models can be serialized through JSON-LD[21] in several formats to ease its readability and interoperability.

The core scope of NGSI is context information capture. The specification combines modelling capabilities to identify context elements with a standardized API for context information management that can be used to support a wide range of smart applications. Context information is flexible enough, with context entities representing the state of any element; an entity could be anything from a temperature sensor in a smart home to a Kubernetes cluster node.

Entities are the base information elements from the specification. An entity is represented by a unique identifier (URI) and a type. For example, a node could be an entity with a unique ID and a type such as "Node". Properties are the attributes or characteristics of an entity. They can include basic data types such as numbers or strings, but can also be more complex, including structured values or even arrays. For instance, a vehicle's speed can be a property with a numeric value. Relationships define how entities are connected to each other or to other resources. A relationship is also a type of property, but instead of a direct value, it points to another entity or an external resource. These three concepts are shared by all linked data specifications, such as the widely popular property graph model supported by the query language Cypher and databases such as Neo4J[22] or Cypher[23].

The combination of extensibility, interoperability, and compatibility of this specification make it ideally suited to be used as the base abstraction for the ACES knowledge model.

[19] https://www.etsi.org/deliver/etsi_gs/CIM/001_099/006/01.01.01_60/gs_CIM006v010101p.pdf
[20] https://www.fiware.org/
[21] https://json-ld.org/learn.html
[22] https://neo4j.com/
[23] https://opencypher.org/

3.1.4 Available datasets

As final data reference points for defining the ACES knowledge model we have evaluated the most important industrial datasets that provide metrics and details about existing distributed computing infrastructure. Ideally we would use sources that are of the same nature of the ACES platform, but its novelty, and high level of heterogeneity make it impossible for any existing source to be taken as-is. Nonetheless, these datasets can provide unvaluable for guiding the design of our platform.

Datasets contain workload information that can be adapted to train and refine the agent models that will constitute the orchestration components of ACES before the full system is prototyped, and actual metrics can be obtained. For this purpose, we have evaluated the most related datasets publicly available. Within the ACES project we are defining a reference EMDC to have a baseline system upon which the whole architecture will be executed. Datasets should describe results taken from an infrastructure that is similar up to some extent to the ACES EMDC we are targeting in the project. To do so, datasets should have information about the state of hardware resources, e.g. memory, storage, CPU, GPUs and other specialized dedicated hardware. The execution platform should be similar to the selected Kubernetes specification that is described in our ACES D2.1 document. Additionally, from a workload point of view, these datasets should contain information about how many resources of each type were available, to characterize the maximum supply, as well as the amounts of each resource requested, and the duration for these resource requests.

We checked several datasets, and the majority of ML-oriented sets are far from the targeted ACES domain. UCI Machine Learning repository[24] provides a collection of several datasets, but none of them are applicable to ACES: The same applies to both KITTI[25] a vision dataset for autonomous driving, and UADETRAC[26], a Real-world multi-object detection and multi-object tracking dataset. The NSL-KDD[27] dataset is more closely related to the ACES domain, as it provides network security traces, but they are not widely applicable to the whole runtime platform.

There are two main sources of datasets that provide highly valuable data for ACES. These are provided by two of the main public cloud providers: Alibaba and Microsoft Azure. We provide a short summary of each dataset repository:

- Alibaba Cluster Trace Program[28]: This program is published by Alibaba Group. By providing cluster trace from real production, the program helps the researchers, students and people who are interested in the field to get better understanding of the characteristics of modern internet data centers (IDC's) and the workloads. So far, four versions of traces have been released: cluster-trace-v2017, cluster-trace-v2018, cluster-trace-gpu-v2020, and cluster-trace-microservices-v2021.
- Microsoft Azure Traces[29]: This repository contains public releases of Microsoft Azure traces for the benefit of the research and academic community. There are currently two classes of traces: VM Traces and Azure Functions Traces.

These datasets provide large-scale metrics with the internal details of resource capabilities and workloads of large-scale datasets. While these are not identical to the ACES platform, they do present important similarities in the type of resources, and contain the dynamics and patterns found

[24] https://archive.ics.uci.edu/
[25] https://www.cvlibs.net/datasets/kitti/
[26] https://detrac-db.rit.albany.edu/
[27] https://www.unb.ca/cic/datasets/nsl.html
[28] https://github.com/alibaba/clusterdata
[29] https://github.com/Azure/AzurePublicDataset

in real user workloads. Therefore, they will be used during the initial stages of the project to aid in the knowledge model design, as well as in the preparation of synthetic datasets to train and tune the models.

The datasets present some differences, but we provide for reference further details of the most recent one from Alibaba (2023, where a mixed CPU-GPU workload is shared):

- The file openb_pod_list_default.csv contains 8152 tasks submitted to the GPU cluster, listing their resource specifications, QoS, phase and creation/deletion/scheduled times.
- The files openb_pod_list_*.csv emphasizes certain types of workloads such as CPU-only tasks, GPU-sharing tasks, etc.

Table 9 Sample dataset from Alibaba 2023 dataset

	name	cpu_milli	memory_mib	num_gpu	gpu_milli	gpu_spec	qos	pod_phase	creation_time
0	openb-pod-0017	88000	327680	8	1000	nan	Burstable	Succeeded	9437497
1	openb-pod-0022	4000	15258	1	220	nan	BE	Running	9679175
2	openb-pod-0035	16000	32768	1	1000	V100M16|V100M32	LS	Running	9967058

The traces provide fine-grained information. For reference, entries are listed as pods, where each entry has following specification:

- cpu_milli: Number of CPUs requested (in milli)
- memory_mib: Main memory requested (in MiB)
- num_gpu: Number of GPUs requested (integers from 0 to 8)
- gpu_milli: Detailed GPU requested for GPU-sharing workloads (i.e., num_gpu==1) (in milli).
- gpu_spec: Required GPU types, For example, nan means no GPU type constraints while V100M16|V100M32 means the task can run on NVIDIA V100 with either 16GB VRAM or 32GB VRAM.
- qos: Quality of Service (e.g., Burstable, Best Effort (BE), Latency Sensitive (LS))
- pod_phrase: Succeeded, Running, Pending, Failed
- creation_time: Timestamp of creation (in seconds)
- deletion_time: Timestamp of deletion (in seconds)
- scheduled_time: Timestamp of being scheduled (in seconds)

3.2 Techniques for knowledge inference and creation

The collected metrics and external accessory datasets contain information that is often too low level to be usable by the ACES components. Agents need well structured, and formatted features that can be high-level enough to enable understanding the runtime behavior of the different services, understanding past performance and enabling autopoietic decision making. In this section we briefly describe the main data processing techniques that will be used in the project as part of the data management flows to feed our knowledge model.

3.2.1 Time series analysis

In general, when a source of data is accessed in a regular manner over time, the information that can be extracted from it is not only limited to the data samples themselves. The relationship between those samples can also contain valuable information. When those samples are structured as a sequence of measurements sorted by the time they were obtained, they form what is called a Time Series.

A time series is a sequence of data points collected and recorded at regular intervals over a period of time. This collection inherently captures the temporal dependencies and patterns within the data, and this temporal dimension holds crucial information for understanding how the underlying process or phenomenon evolves. When data are collected from a monitoring environment, this data structure emerges naturally, as the measurements are taken at regular intervals. One of the most common objectives of system monitoring is to detect anomalies in the system's behavior, and time series analysis is a powerful tool for this task. Also, time series analysis is a fundamental tool for forecasting future values of time series, which is a key component of many predictive systems.

Time series usually describe the temporal evolution of a single variable; in that case the series is known as univariate. Although it is possible to combine several series into one that represents several variables, in that case they are considered multivariate series. This approach not only creates a much more complex structure, but also can be used to model the temporal interdependency between the variables. The behavior of some variables may have a big impact on other variables; for example, the temperature of a room will have a great correlation with the temperature of a device located in that room. In the case of this project, several variables are monitored, for example, Prometheus will be used to collect metric about CPU usage, memory usage, network traffic, etc. All these metrics are collected at the same time, so they can be combined into a multivariate time series.

Time series can be represented as a simple sequence of samples but are usually represented as a sequence of tuples (t, x), where t is the time at which the sample was taken, and x is the value of the sample. With this representation, the information stored in the series is enhanced, as the temporal dimension is explicitly represented. This allows to perform operations on the time dimension, for example, to calculate the time difference between two samples. In addition, it facilitates the identification of missing samples, which is a common problem in time series data. The time difference between two samples, which is usually constant, is called the sampling period or, in some cases the sampling frequency, calculated as the inverse of the sampling period.

From a mathematical point of view, a time series is a stochastic process, which is a collection of random variables indexed by time, and it can be represented as a function of time. The time series can be represented as a function of time, $X(t)$, where t is the time index and X is a random variable. In practical cases, the time series is not a continuous function, but a discrete function, as the samples are taken at discrete time intervals. However, most of the time series and continuous function analysis techniques can be applied to discrete time series with little or no modification.

3.2.1.1 Applications of Time Series Analysis

Time series analysis is a very broad field, and it has many applications in different domains. In this section, some of the most common applications of time series analysis are presented.

- **Forecasting**: Forecasting is the most common application of time series analysis. The goal of forecasting is to predict future values of the time series. The forecasting techniques use the information contained in the time series to predict the future values of the time series.

- **Anomaly detection**: The goal of anomaly detection is to detect abnormal behavior in the time series. For example, in the case of systems monitoring, it is very important to be able to detect unusual function of the system, as it allows one to detect problems in the system before they become critical.
- **Clustering**: The goal of clustering is to group the time series into different clusters. Each cluster contains time series that are similar to each other in some way, usually not known in advance. When applied to monitoring, it allows to group different components of the system that behave in a similar way. Once these groups are identified, it is possible, for example, to detect problems in some components of the system by analyzing the behavior of other components in the same group.
- **Visualization**: Although visualization is not always considered an application or a final goal, it is a very important part of time series analysis. Understanding how a system behaves over time is not an easy task, and time series visualization can be a very powerful tool for this.
- **Summarization**: The goal of summarization is to condense the information contained in the time series into a smaller representation. This is useful when the time series is very large, and it is not possible to analyze it in its entirety or when it is necessary to compare several time series.

3.2.1.2 Time Series enhancement techniques

Time series are powerful tools for analyzing the temporal evolution of a system, but they are not always easy to interpret. The information contained in a time series is not always easy to extract, and it is often necessary to apply some enhancement techniques to make the information more accessible. In this section, some of the most common enhancement techniques are presented.

Smoothing: In some cases, the time series may contain a lot of noise, which makes it difficult to extract the information contained in the series. Or the sampling frequency may be very high, and the number of samples can shadow the information contained in the series. In these cases, it is useful to apply some smoothing techniques to reduce the noise and make the information more accessible. There are many smoothing techniques, but the most common ones are moving average and exponential smoothing. The result of applying a smoothing technique is a new time series in which high-frequency components have been removed.

Detrending: The trend of a time series refers to the long-term movement or pattern in the data over time. It represents the underlying, gradual changes in the data that occur due to various factors such as component degradation, demographic changes, etc. The trend is usually not of interest, as it does not contain much information about the system behavior. In fact, it is usually considered a source of noise because it can hide the information contained in the series. In those cases, it is useful to remove the trend from the series. The result of removing the trend is a new time series in which the long-term movement has been removed. To remove the trend, it is necessary to estimate the trend function and then subtract it from the original series. The trend function can be estimated using different techniques; the most common ones are linear regression and moving average.

Seasonal Decomposition: Seasonality in a time series refers to regular and predictable patterns or fluctuations in the data that occur at specific intervals of time. These patterns often repeat over a short-term period, such as a day, week, month, or a specific season, and are typically associated with external factors or recurring events. For example, it is common to observe a weekly seasonality when working with systems with users. The behavior of those users is usually different during the week and during the weekend. As with the trend, the seasonality is something predictable, that once it is identified, it can be removed from the series. In some cases, a time series can have several seasonal components; considering again human behavior, it usually presents a weekly seasonality, but also a daily seasonality as the lunch time or resting hours are repeated every day. To identify these seasonal components, most common techniques involve analyzing the data in the frequency (or

spectral) domain to identify those periodic components which contribute to seasonality. The most common technique for this purpose is the Fast Fourier Transform (FFT), which can decompose the time series into its constituent frequencies.

Time Series Decomposition: The two previous techniques are usually presented together to create a set of three functions: one for the trend component, one for the seasonal component and one for the residual component. The residual component is the part of the series that is not explained by the trend or the seasonal component. This residual component is usually considered the most interesting part of the series as it contains the information about the system behavior that is not directly explained by the previous data.

3.2.1.3 Combining Time Series Data

In the context of this project, various variables are monitored, for instance, Prometheus will be employed to gather metrics on CPU usage, memory usage, network traffic, etc. Each of these metrics generates a time series for its respective variable, and they can collectively form a multivariate time series. When all the series share the same sampling frequency, the resultant multivariate time series is called a regular multivariate time series. However, it is possible that the sampling frequency varies for each variable, e.g. some variables being obtained every second and others every minute. In those cases, the resulting multivariate time series is called an irregular multivariate time series. To work with these irregular series, it is necessary to process the sampling frequency of each variable to create a regular multivariate series.

Resampling is the process of altering the sampling frequency of a time series, and it can either increase or decrease the sampling frequency. In the aforementioned case, resampling is one of the processes used to create a regular multivariate time series. To achieve a target frequency some series will need to increase their sampling frequency, whereas other will need to decrease it. Increasing the sampling frequency is called upsampling, while decreasing it is called downsampling. Downsampling is typically a straightforward process, involving the removal of some samples from the series. However, upsampling is a more complex process that involves creating new samples to fill the gaps between the original samples. This process of generating new samples is known as interpolation and can also be employed to fill missing samples in a time series.

Choosing the target sampling frequency involves considering the trade-off between upsampling and downsampling. When upsampling is performed, the new samples are synthetic estimates and may not be as accurate as the original samples. Conversely, when downsampling is carried out, some samples are removed, resulting in the loss of some information.

3.2.1.4 Other Cleaning and Preprocessing Techniques

Time series analytics can be a very powerful tool for analyzing the behavior of a system, but usually it is quite sensitive to noise and other artifacts like missing samples or outliers. In addition, it is not always easy to extract the information contained in the series. In some cases, it is necessary to apply some preprocessing techniques to make the information more accessible.

Outlier Detection: Outliers are data points that are significantly different from the other data points in the series. They are usually caused by measurement errors or sensor malfunctions, but they can also be caused by some unusual behavior in the system. It usually appears as high frequency components in the series, as isolated samples.

Error Correction Methods: The problems in sensors or acquisition processes can cause errors in the data. Some examples of these errors are duplicated or delayed samples. These errors may be difficult

to detect, but they can cause problems in the analysis of the series or biased results. Statistical analysis or comparison between different sources of data can be used to detect these errors.

Data Normalization: it is a common technique used to scale the data to a fixed range. When combining several time series into a multivariate time series, it is usually necessary to normalize the data to a fixed range as the different series may contain information in different scales. For example, the CPU usage is usually a percentage, while the number of requests per second of a service can be an arbitrary high number.

Data Imputation: missing samples in a series can be caused by several factors, like network congestion, services failures, server maintenance, etc. Data imputation is the process of filling the gaps produced by missing samples. Interpolation or mean imputation (replace the missing values with the mean value of the series) are some of the most common techniques, however, the analysis of the series can be used to create more accurate estimations based, for example, in the trend or the seasonality.

Feature Engineering: The date and time information contained in a time series can be used to extract additional information. For example, the day of the week, the hour of the day, the month of the year, etc. That kind of features are usually transformed to periodical signals, as they are cyclical in nature. For example, the hour of the day can be represented as a signal that repeats every day, period 24. This transformation helps to relate the samples belonging to the beginning of the cycle to the ones of its end, e.g., samples collected at 23:00 and 00:30.

3.2.2 Dependency Analysis

Dependency analysis is a type of data analytics that aims to study the relationships between variables, events, and observations collected from a system. Understanding the relationships among this data makes it possible to discover and identify connections between components or processes in the actual system being analyzed. These relationships or dependencies may be intentional and result from the architecture or design of the system, but as systems grow in complexity, unforeseen internal dependencies often emerge. Analyzing these dependencies is useful in various ways. The first utility is diagnostic, meaning a reactive approach that helps understand the possible causes of an event or the consequences that event has had on other components of the system. Taking this approach a step further, dependency analyses can be conducted proactively, leading to fault prediction tools that anticipate the effects of an event before it occurs, enabling preventive measures to avoid or at least mitigate its impact.

Dependency analysis is a broad field that encompasses many different techniques and approaches. In this section, some of the most common techniques are presented.

3.2.2.1 Root Cause Analysis

Root cause analysis (RCA) is a method used to identify the root causes of faults or problems. Some event is considered a root cause if its presence or absence directly and specifically results in the occurrence of the problem. RCA seeks to identify the point of origin of a problem studying the chain of events that lead to the problem. RCA can be applied systematically following a methodology to identify and address the fundamental causes of faults or problems. Within this methodology, a specific event is deemed a root cause if its presence or absence directly and unequivocally leads to the occurrence of the problem at hand. RCA goes beyond addressing surface-level issues by delving into the intricate details of the chain of events that culminate in the manifestation of the problem. This analytical approach aims to pinpoint the exact origin point of the problem, unravelling the underlying

factors and circumstances that contribute to its development. The steps involved in the RCA methodology are as follows:

- **Problem statement**: define and describe the events, failures or situations that will be studied.
- **Data collection**: gather all data and events related to create a timeline or chronology.
- **Estimate the impacts**: estimate through historical correlations, differentiation, etc. the impact or effects of the different events in the subsequent. This step should distinguish causal factors, and non-causal factors.
- **Causal graphing**: Finally, using the sequences of events from previous step, a subsequence of key events that explain the problem should be obtained and converted into a chain event graph.

3.2.2.2 Graphs and Event Representation

Among all the data structures or information formats used in this field, graphs stand out above the rest. A graph is a data structure that models information to represent relationships between different nodes. These relationships are directly mapped to dependencies, and the nodes can represent variables, events, or the components themselves. This abstraction allows this, a priori generic, structure to be used in this field to represent information, and even more importantly, enables the use of graph-based algorithms for dependency analysis.

Graphs are an abstract data structure and can be employed in multiple ways to represent the same information. For example, a system can be modeled by placing its components as nodes and representing communication between components as edges between the nodes. Another option could be to model the system as a state machine, with edges representing transitions and actions taken in the system's components. Despite the mentioned flexibility to represent the same information in different ways, there are some graph structures or types that are more commonly used than others due to certain properties they possess. The most important ones are briefly described below.

Directed Acyclic Graph (DAG): A DAG is a type of graph that consists of nodes connected by directed edges, and it does not contain cycles. DAGs are commonly used to represent dependencies and relationships in systems where certain actions must occur before others. DAGs can illustrate relationships between components or tasks where one must precede another. In this context, nodes represent components or tasks, and directed edges indicate dependencies. DAGs are valuable for visually mapping out dependencies and ensuring that there are no circular relationships. In the context of RCA, a DAG could illustrate the causal relationships among various factors contributing to a problem, emphasizing the sequence and hierarchy of events leading to the identified root cause.

Event-Driven Graph: are utilized to model systems where actions or occurrences trigger subsequent events. Nodes in this graph represent events, and directed edges signify the cause-and-effect relationships between events. This type of graph is useful for understanding the chronological order and dependencies of events within a system. The most common application of these graphs is the Chain Event Graph, described later in this section.

Time-Based Graph: this type of graph focusses on capturing temporal relationships within a system. Nodes represent events or states, and edges denote the temporal order between them, adding a new dimension to the data structure. This temporal dimension is usually represented as a property in the edges. This type of graph is beneficial for analyzing how the timing of events influences the occurrence of problems.

3.2.2.3 Chain Event Graphs

Chain Event Graphs area graph graphical model for representing and analyzing causal relationships within a system. These graphs are characterized by a sequential arrangement of events, forming a linear chain that visually depicts the cause-and-effect dependencies among them. Each node in the graph represents a specific event or state, while directed edges signify the chronological order in which events unfold. These nodes in the chains may have several predecessor and descendants, but in most cases the chains converge in a single one that contains the event of interest for the analysis. In many cases, as this aggrupation of chains do not create cycles, they can be modelled as a DAG.

Chain Event Graphs are valuable tools for dependency analysis, allowing to identify critical dependencies and understand the intricate relationships between events. By visually examining the connections between events, decision-makers can gain insights into the impact of changes or interventions on the overall sequence of events. This type of visualization is usually both a result and a tool in the RCA processes.

3.2.2.4 Call Graph Analysis

Similar to the representation of events in graphs, interactions between different components can also be modelled in a graph. These graphs are commonly known as call graphs, and they are used to represent the interactions between the different components of a system. In this context, nodes represent different components, services, or subsystems, while edges represent the interactions between them. These interactions typically represent method invocations, data transmission, and various forms of communication. In most cases, these graphs are time-based graphs, as this modelling approach enables the representation of the chronological sequence of interactions between the different components.

3.2.3 Network flow statistics processing

The ACES network switches will compute fine-grained, flow-based metrics, *per packet*, directly in the data plane. As explained in Deliverable 2.1, when deployed in an EMDC edge at Terabit traffic speeds, conventional server-based solutions can only monitor *a small subset of traffic* for its downstream applications, as they are limited to a few Gbps packet processing at best. Network traffic needs thus to be (heavily) sampled to meet the capabilities of the monitoring server. Observing and computing in-network statistics *over all network traffic* (see the details on network metrics in Table 7) in the network switch data plane makes the ACES monitor records richer than the sampling-based records generated by traditional systems, enabling new and improved network monitoring applications.

We can divide the process of moving from network traffic data to knowledge into three parts: packet processing, statistics computation, and statistics analysis. **Packet processing** is the task at which a packet switch excels. An incoming packet is parsed according to the specific network protocol (e.g., extract the IP destination header). The parser extracts fields from the packet headers, making them available for subsequent processing. Next, the extracted packet header fields are used as keys for the switch lookup tables (e.g., the forwarding table). Finally, once the appropriate entry is identified in the tables, the switch performs an action (e.g., forwarding the packet to a specific port).

In ACES, we will develop specialized actions to extract telemetry data. We can consider these data along two axes:

Flow type. The ACES network switch monitor will compute metrics for multiple flow keys. Currently, we are considering four types of keys: *[MAC src, IP src]*, *[IP src]*, *[IP src, IP dst]*, and *[5-tuple]*.

Flow atoms. The ACES switch stores telemetry data as "flow atoms". These are specialized counters for a specific flow key. Currently, we consider three flow atoms: number of packets, number of bytes, and squared number of bytes.

These atoms are maintained in the switches' stateful memory and are used to **compute statistics**, the second step in moving from network data to knowledge. For generality, the ACES switch will compute a diverse set of statistics of two types: unidirectional (1D), tracking outbound traffic, and bidirectional (2D), considering both inbound and outbound traffic. The 1D flow statistics include weight, mean, standard deviation, and time intervals. The 2D statistics include magnitude, radius, approximate covariance, and correlation coefficient.

An important observation is that computing these statistics can be performed in a streaming fashion, per-packet – much aligned with the computational model of a switch pipeline. Indeed, the pipeline stages of a switch allow for performing basic arithmetic and logical operations per packet, storing the results in stateful memory when needed. In ACES, we instrument the switch pipeline to compute the statistical features of network flows. However, to maintain Terabit throughputs, the computational model of a switch is limited, and many calculations resort to approximations. There is, therefore, a trade-off between the ability to compute statistics over all packets and the potential loss of precision of approximated estimates.

The final step for knowledge creation is **statistics analysis**. In ACES, we will employ AI/ML techniques to extract knowledge from the flow statistics. For example, the anomaly-based intrusion detection systems developed in ACES will use network flow statistics computed in the switch as input features to an ML processing pipeline. We will investigate autoencoders, specifically, as these models can be trained to mimic (reconstruct) network traffic patterns [10]. The discrepancy between this input and the reconstructed output can serve as a measure of anomaly.

3.2.4 Feature extraction and modelling techniques for security and privacy

As presented in D2.1, ACES offers a comprehensive security solution to safeguard its services and data from potential cyberattacks. This solution involves various techniques to extract features, model system behaviours, and detect cyberattacks, as summarized below.

3.2.4.1 Anomaly detection in EMDCs

For container and node security, multiple metrics discussed in Section 2.3 - such as system calls, CPU, GPU, memory, storage, and network metrics - can be used to profile the normal behavior of the system and detect anomalies that may be caused by cyberattacks. These metrics can be collected and processed as time series data, with time series analysis typically applied for anomaly detection or clustering (cf., Section 3.2.1.1). Beyond time series or statistical analysis, advanced ML models like Decision Trees [11], K-NN, Random Forest, and AutoEncoders [12][13] are utilized for dynamic anomaly detection. These ML-based techniques learn and compare the patterns of system calls triggered and the CPU, memory, and network usage generated by benign and malicious workloads. For instance, time series data of system calls can be transformed into bags and sequences of system calls before using supervised anomaly detection methods to identify potential attacks [14].

3.2.4.2. Security for ML

The ACES agents will be supported by several types of ML models based on the training data sources we have described in this document. For ML security, there are various techniques to analyze ML model parameters (weights and biases) to detect poisoned models. Most techniques convert the tensors of model parameters into vectors and process them as time series data. For example, Fereidooni et al. transform model parameter vectors into the frequency domain and use frequency analysis techniques to identify poisoned models [16]. Additionally, several techniques to discern differences between poisoned and benign models include Euclidean and cosine distance measures [15], data distribution analysis [17], and frequency domain analysis. Typical ML algorithms used for clustering and detecting poisoned models are, for instance, K-Means [18] and HDBSCAN [15][17].

4. ACES Knowledge Model

This section presents the knowledge model defined in ACES to characterize all the context information relevant for the ACES agents to reach decisions on what actions should be started by the system. The section provides both the description of the main parts of the knowledge model, as well as the relationship of this model with the agents that will make use of the information.

4.1 Knowledge Model description

The model aims to capture a comprehensive set of information that will be instrumental as features in these machine learning models, enhancing the decision-making and predictive capabilities of the ACES agents.

Knowledge will be structured within a knowledge graph, allowing for complex relationships and dependencies to be represented and queried efficiently. To facilitate interoperability and extensibility, our abstractions will adhere to the NGSI-LD standard. This alignment not only facilitates the integration of ACES elements with other European Data Spaces but also enhances the scalability and adaptability of our approach.

These abstractions are being defined based on the following data sources that have been discussed in the previous sections.

The information model is built on the foundation of the ACES context and requirements, which have been meticulously documented in Work Package 2 (WP2), with specific emphasis on Deliverable 2.1 concerning the ACES architecture. This architecture defines the overarching structure and guiding principles for the creation of management agents capable of overseeing complex edge computing environments.

We incorporate insights taken from industry-leading datasets as the ones discussed in Section 3.1.4. These datasets show real configurations and scenarios, as well as highlight key characteristics from these environments that must be captured for an effective management of ACES environments.

Our model also takes into account metadata derived from prevalent microservice configurations and complex deployments, including those found in widely used Docker and Helm repositories. This data is critical for understanding the types of dependencies, relationships, and constraints that the ACES agents must manage. Such insights are imperative for the autopoietic functions of the agents, enabling them to adapt and evolve within their operational ecosystem.

The model also integrates runtime information from the supply-side, such as the runtime state of services, and the information captured by monitoring metrics, as detailed in Section 2. These metrics provide real-time insights into the performance and state of the micro data centre and the running services, which will provide a substantial percentage of the features that ACES agents employ for reasoning.

4.1.1 Modelling Challenges

The ACES platform targets applications adapted to the specific capabilities of the edge-cloud continuum and proposes an innovative hardware platform to run its functionality (the EMDC). We briefly discuss some of the unique characteristics that we have considered.

4.1.1.1 EMDC Resource Pool

Additionally to the nodes in an EMDC, we consider a pool of resources that presents an innovation to the current definitions of the edge continuum. This means that besides the processing capabilities in a node (that is a constitution of multiple resources), single resources can be requested for pod processing. This pool of resources is part of the EMDC and can be consulted by the edge(-cloud) management as requested. Such a pool mainly prevents resource limits, increased latencies, and stability of the performance of other pods, as their assigned resources are not tapped. Currently, the Compute Express Link (CXL)[30] is being implemented in CPUs (Intel, AMD), in memory and storage (Samsung) and the PCIe switches are expected in 2025. Besides the hardware development, the biggest challenge currently is related to orchestration and how the network infrastructure can be incorporated into these pools of resources.

4.1.1.2 Application Types

For the different services, we can identify three application types that come with diverse requirements in their response time.

- The long-running applications (LRAs) instantiate long-standing pods to enable iterative computations in memory or unceasing request-response. LRAs include processing frameworks (e.g., Apache Spark[31], Flink[32]), latency-sensitive database applications (e.g., HBase[33] and MongoDB[34]), and data-intensive in-memory computing frameworks (e.g., TensorFlow[35]).
- Batch processing is typically used when you have a large amount of data that needs to be processed all at once, and when the results of that processing can be stored and used later. Data is typically processed on a schedule or at regular intervals. There are two types of batch processing: Regular returning requests, and opportunistic requests with little to no SLA (Service Level Agreement).
- Stream processing also deals with large volumes of data, but the data needs to be processed in real-time (e.g. Apache Storm[36], Kafka streams[37]).

Future workloads will become even more complex with LRAs, batches, and stream processes being interconnected. Therefore, it will be challenging to categorize an application and tune its agents accordingly.

4.1.1.3 Relationships among Pods

Applications will be deployed in the runtime platform as pods. These pods can have several relations with each other. There can be different needs, e.g., that they need to be processed in parallel or that they depend on each other. Additionally, if one pod is too slow, the current system creates more pods to reach the given response times of the specific services. Currently, these relationships are not used

[30] https://www.computeexpresslink.org/
[31] https://spark.apache.org/
[32] https://flink.apache.org/
[33] https://hbase.apache.org/
[34] https://www.mongodb.com/
[35] https://www.tensorflow.org/
[36] https://storm.apache.org/
[37] https://kafka.apache.org

in the scheduler and orchestration optimization. For example, placing interacting services closer together can significantly enhance their performance) 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 pods that are heavily dependent on a database, it is best to place them near the database to reduce latency and improve overall performance.

4.1.2 Base concepts

Our knowledge model is built on top of the NGSI-LD framework, utilizing its foundational abstractions—Entity, Relationship, Property, and Value—to construct a comprehensive knowledge graph that encapsulates the multifaceted nature of service orchestration and infrastructure management.

The knowledge graph is also captured using the JSON-LD format, ensuring a standardized and interoperable representation of information.

Entities within our model serve as the primary abstraction, characterizing the different aspects of supply and demand. They encapsulate detailed descriptions of the runtime services executing at the ACES platform, including the logical definition of their constituent components such as pods, replicas, and the various other elements that necessitate accommodation within the infrastructure.

Parallel to the service descriptions, our model delineates the supply aspect, providing a structured description of the environment. This encompasses the EMDC, detailing the available hardware resources—CPU, memory, storage, and networking components. Both aggregated and disaggregated resources are represented, reflecting the actual state of the infrastructure. The runtime orchestration platform, Kubernetes, is depicted through entities that describe the managing nodes and the deployed pods.

Monitoring information forms the third pillar of our model, encompassing metrics that are intimately related to the supply elements. These metrics capture the performance and utilization of hardware and software resources, providing insights into the infrastructure's operational status. Events that record the interactions and invocations of the runtime-deployed microservices enrich the model, offering a dynamic perspective of system behavior and service consumption.

These three models are directly related. The services described in the supply model will be linked to their real instantiations in nodes from the demand model, with individual instances of the execution services being linkable to all their requirements and predefined constraints. Finally, the runtime model will complete information about every element of the supply model, providing historical and up to date views of their state.

4.1.3 Supply model

The provided diagram shows the key entities of the supply model and their interrelationships, which capture the ACES services runtime execution platform.

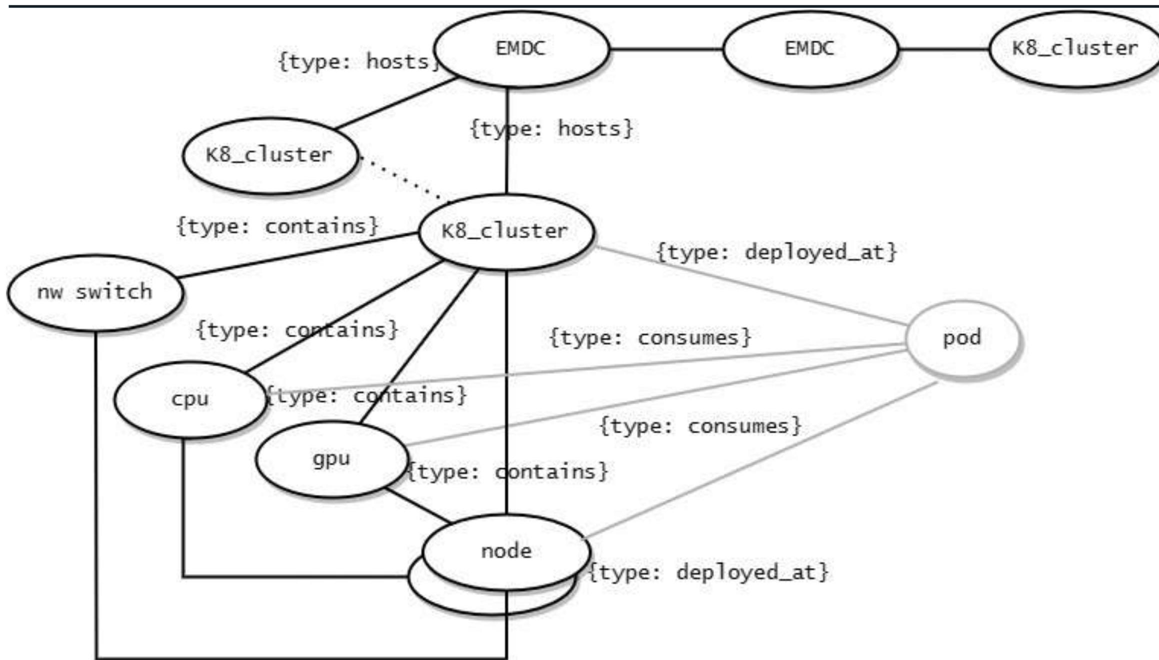

Figure 2 ACES sample supply model

The top-most level element is the **EMDC** (Enterprise Micro Data Centers), which serves as the hosting ground for the Kubernetes clusters. The EMDC represents the hardware platform that contains multiple entities representing the hardware that will be used to deploy and run services. As the ACES architecture described In D2.1 describes, there can be multiple EMDCs that form the ACES execution infrastructure. Key properties of an EMDC might include:

- **id**: A unique identifier, such as **"emdc001"**.
- **location**: The geographical or network location, for instance, **"DataCenterNorth"**.

The EMDC hosts multiple **K8_cluster** entities, each representing a Kubernetes cluster. The **hosts** relationship makes these explicit in the model. A Kubernetes cluster manages a set of node machines for running containerized applications. Potential properties for a K8_cluster include:

- **clusterId**: A unique identifier, like **"cluster-01"**.
- **nodeCount**: The number of nodes in the cluster, e.g. **"15"**.

A **node** is a worker machine in Kubernetes that can run Pods for ACES services. It contains the services necessary to run Pods and is managed by the master components. Examples of node properties are:

- **nodeId**: The unique identifier, such as **"node-1234"**.
- **status**: Current status, e.g. **"Available"** or **"Unavailable"**.

ACES services, deployed as pods will require access and consumption of multiple types of resources: **cpu**, **gpu, memory,** and **storage**. Each type of resources is modelled as a separate entity. Unlike traditional hardware architectures, the ACES EMDC will provide a mesh of disaggregated resources, which is represented in our modeled with the **contains** type of relationship being defined not only from a node to one of these entities, as it would happen in a classical system, but also by the possibility of these links occuring directly from e.g a gpu to the EMDC. Each of these entities will have its set of unique potential properties, although we provide some examples among them:

- **cores**: The number of cores available, e.g., **"8 cores"** for CPU.
- **model**: The model of the processor, such as **"Intel Xeon E5-2670"** for CPU or **"NVIDIA Tesla V100"** for GPU.

These entities represent every aspect of the demand, including their topology, and current state. This model is completed with the key entity that maps between supply and demand, the **pod** (we use the Kubernetes term in this case, as pods are the smallest deployable units of computing that can be created and managed in Kubernetes). Each pod runs a single instance of a given service. A pod runs on a k8_cluster, as shown with the **deployed_at** type of link. For a pod, relevant properties include:

- **podId**: A unique identifier inside the cluster, such as **"pod-5678"**.
- **ServiceId:** a reference to the service definition in the supply model where the full information about the service can be extracted.

Consumes is key to capture the resource usage for each pod on the. This edge is annotated with properties that qualify the amount of that resource currently being allocated to that node, for instance, **{"cpu": "250m", "memory": "512Mi"}**.

This structured supply model provides a clear and comprehensive view of the available resources and their utilization, which is essential for maintaining the desired performance and efficiency of the ACES platform's services.

4.1.4 Demand model

Kubernetes applications are structured as services containing pods, with explicit dependencies. Helm charts build upon these elements and provide a higher level services view, focusing on aspects like dependencies and compositions. The ACES demand model needs to support multiple types of applications that will be deployed under this model: Long-Running Applications (LRAs), batch processing applications, and stream processing applications. Each type presents unique aspects in terms of operation: some are designed to execute a task and then terminate, while others are intended for continuous operation. These applications are also defined by their performance objectives, such as completion times and deadlines for batch processes, or service times and application latency for continuous operations.

In defining the requirements for these applications, we consider additional goals and constraints that are specific to the services and charts. These requirements form the basis of the demand model (Figure 3). This approach ensures that each application's needs are accurately captured and addressed in the Kubernetes environment.

The demand model is crafted to encapsulate the objectives and constraints integral to the ACES platform's functionality, performance, and operational correctness. These goals serve as inputs for reasoning agents, ensuring the platform's autopoietic behavior aligns seamlessly with the intended outcomes.

The core entity of the demand is the **service**. This entity represents a discrete unit of functionality—be it a microservice, Function-as-a-Service (FaaS), or Database-as-a-Service (DBaaS). It is important to distinguish between this general notion of a service and a Kubernetes service, which, although similar, entails additional specificities related to deployment and networking. Potential properties for a service entity might include:

- **id**: A unique identifier for the service, such as **"energy auction service"**.
- **version**: The current version of the service, for example, **"1.4.3"**.
- **provider**: The entity providing the service, e.g **"IPTO"**.

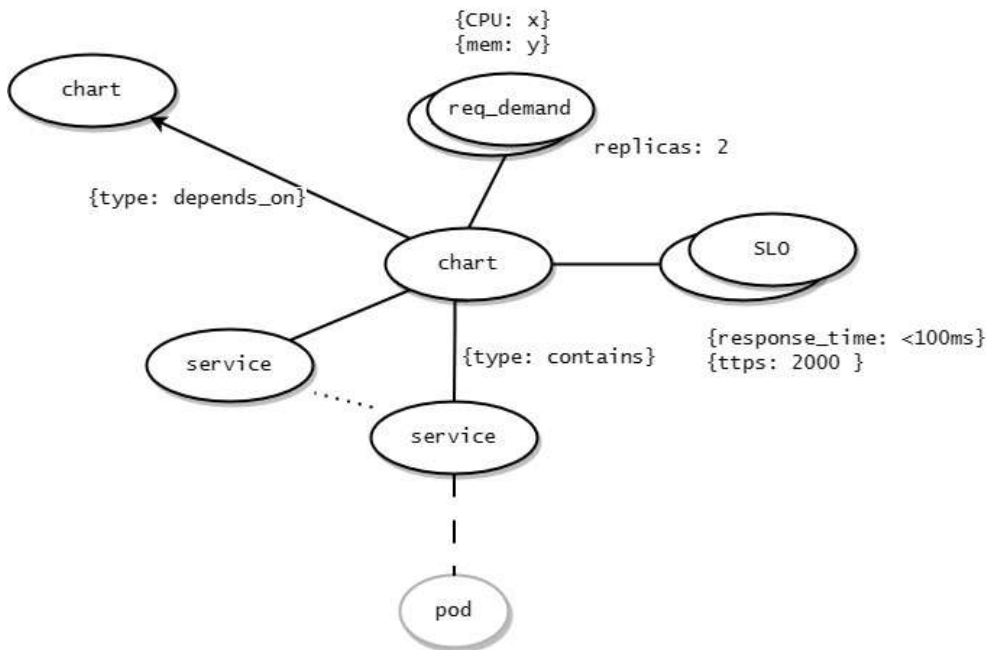

Figure 3 ACES Sample demand model

Services are not standalone; they frequently amalgamate into higher-level constructs for more complex functionality. We refer to these assemblies as **charts**, resonating with the Helm terminology, the prevalent tool for defining and deploying such groupings. A chart encapsulates a collection of services, managing their deployment as a coherent unit. Relationships between charts, often encapsulating composition and dependencies, are represented as edges in our model, with the type **depends**. Chart entity properties may include:

- **id**: A unique identifier for the chart, such as **""**.
- **version**: The release version of the chart, for instance, **"2.4.0"**.
- **provider**: The entity providing the whole chart, will be the same of its services.

Services within the supply model are constrained by definitive **requirements**. These articulate the resource commitments necessary for a service's operation, whether through complete dedication or reserved capacities. Properties for a requirement entity could encompass:

- **resourceType**: The kind of resource required, like **"CPU"** or **"Memory"**.
- **quantity**: The amount of the resource needed, for example, **"2048MiB"** for memory.
- **reservation**: A boolean indicating whether the resource is exclusively reserved, e.g.**true**.

Lastly, the model integrates **SLO** (Service Level Objectives) entities, which express the behavioral goals of services during execution. These SLOs detail the non-functional aspects, such as performance thresholds or reliability targets, that the services are expected to uphold. Properties for an SLO entity might include:

- **metric**: The performance metric it pertains to, such as **"latency"**.
- **target**: The desired threshold for the metric, possibly **"100ms"**.
- **reliability**: A measure of uptime or error rate, for example, **"99.9%"**.

Together, these entities and their interconnections construct a demand model that guides the reasoning agents in orchestrating a dynamic, responsive, and efficient system operation.

4.1.5 Runtime model

The runtime model links the supply model elements that characterize the ACES platform resources, with the demand model describing the functionality that needs to be deployed. In order to ensure that everything works according to the set requirements, it is necessary for the knowledge model to be able to capture at each point in time what is the exact state for every supply model. This way, ACES reasoning agents will have the ability to explore the past and present of the environment and take informed decisions.

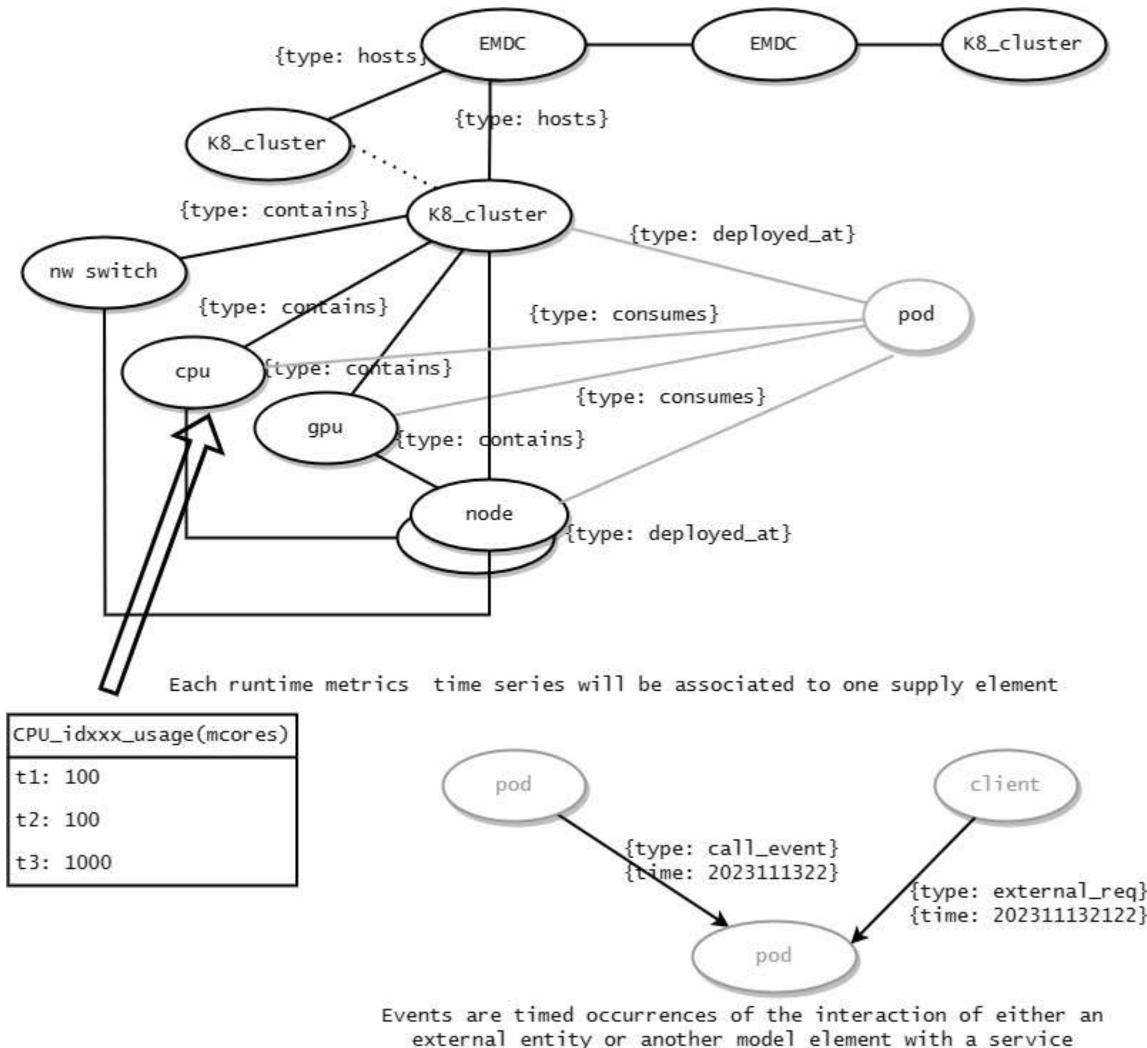

Figure 4 ACES sample runtime model

The runtime model is composed of two entities: metrics and events. Metrics are time series that are associated to a single element of the supply model. Metrics have a name, and each of them a timestamp of the obtained measurement time. This way, the whole set of metrics constitutes a time series.

Events are timed occurrences of interactions on the runtime elements, in particular the deployed pods that instantiate demand services. Events also have a timestamp, and they might originate from another element of the runtime model (e.g. from another pod, constituting a service call), or from some external element such as a client. These are individual entries, but agents can process them into more aggregate metrics to aid in the decisions, such as workload aggregated statistics.

4.2 ACES agent types

The knowledge base of ACES will be used by reasoning entities that we refer to as agents. In this subsection we provide a short overview of the main characteristics of a system for decision making based on agents using the Agent-Based Modelling (ABM) approach. The information presented here is complemented with Section 4 of D4.2, where we present specific agents and algorithms to handle workload management problems in ACES.

In ABM, a swarm consists of swarm members that can be modelled as agents. They follow local rules, interact with the environment, communicate with other agents, and react on local information [20].

Wilensky and Rand [19] give the following guidelines when to use ABM:

- **Medium number of agents**: several dozen up to about 100000 agents. In our use case, we model up to several thousand agents, typically pods and resources.
- **Heterogeneity**: in ABM, agents can be as heterogeneous as necessary. Thus, in ABM, we can model different pods and resource types as agents.
- **Local and complex interactions**: as used in swarm intelligence can be depicted in ABM.
- **Rich environments with agent-like local rules**: This can be used to, e.g., model complex node queue manipulations in our case.
- **Time**: ABM is a model of process that fits to our job-shop scheduling problem.
- **Adaptation**: almost no other method can model adaptivity of individual entities well. In ABM, agents' actions and decisions depend on past actions and current information, i.e., agents can learn. This fits the swarm model very well.

When shaping the edge continuum to an agent-based system, we analyze a group of possible swarm agents and their attributes. In this context, we need to determine the eligibility of an entity to serve as a member of the EMDC swarm [21]. The swarm can exhibit homogeneity (with all agents being of the same kind, like numerous pods) or heterogeneity (comprising agents of various types, such as pods and resources). For an entity to qualify as a swarm member, it should possess the capacity to effectively function within a swarm. This entails the presence of a significant number of other swarm members (for instance, a single instance of an FPGA, existing in isolation, would not make a suitable swarm member). Additionally, the entity should exhibit an appropriate degree of abstraction to facilitate modelling, possess the capability to sense dynamic information from the immediate environment, respond to information originating from the local vicinity (such as making decisions), and be logically coherent and comprehensible, fostering trust in the proposed solution [20].

Our agent-based approach introduces two distinct types of swarm agents: demand swarm agents and supply swarm agents. These agents collaborate within an EMDC environment, orchestrating processes such as pod placement, storage management, and caching optimization. The model for the problem consists of an edge continuum with resources, queues, pods, and processes (following subsections are adapted inputs from Schranz et al. [22]).

4.2.1 Demand Swarm Agents

An application is split to a set of services S, that are represented as a set of related pods $P^s = \{p_1, p_2, \ldots\}$ with s as the specific service. Each service s is defined by a compilation of resources R^s which prescribes the processing steps necessary to compute the individual pods. The pod p_j^s can choose which of the suitable nodes N_i^n to use for each necessary process step P^r.

4.2.2 Supply Swarm Agents

The EMDC E contains several sets of nodes or nodes, consisting of different types of resources $N^r = \{N_1^r, N_2^r, \dots\}$, where r is/are the resource type(s). A node with different resources presents a typical EMDC node, whereas a node with a single resource presents, e.g., a CPU that is part of a pool of resources. In the course of this project, we consider multiple types of resources along with their respective capacities: CPU, FPGA, RAM, and NVMe. Each resource N_i^r has a queue Q_i^r.

4.2.3 Orchestration of Swarm Agents with ML

One key element of the overarching ACES architecture revolves around the orchestration of these demand/supply agents. This orchestration process will be executed through swarm algorithms.

Each individual swarm agent will adhere to specific policies and conform to a general behavior pattern established by the chosen swarm algorithm. However, it is worth noting that every swarm algorithm relies on hyperparameters that fine-tune various aspects of the resulting coalition's behavior [23]. More specifically: 1) hormone algorithms are contingent on hyperparameters that govern i) the quantity of generated hormones, ii) the rate of hormone evaporation, iii) the mobility of hormones, and iv) the intensity of hormone attraction; similarly, 2) ant algorithms involve hyperparameters related to i) the influence of hormones, ii) the rate of updates, and iii) the evaporation rate. Hyperparameters are typically chosen using trial-and-error methods, random/grid searches, and/or heuristics [24]. Once these values are established, it is uncommon to modify them during the execution of the swarm algorithm.

In a novel approach, ACES selects hyperparameters using autonomous ML techniques which also allows for potential real-time updates, enabling the coalition's behavior to adapt to significant environmental changes. More specifically, Bayesian learning [25] and Reinforcement Learning [26] tools will be employed and tested for this purpose. These two are experiment-driven approaches that efficiently explore the hyperparameter space by monitoring the system's KPIs. They offer efficiency as i) they are automated and ii) yield satisfactory results with a limited number of iterations [27]. Additionally, once a suitable set of hyperparameters is identified, these ML tools can swiftly self-adapt to environmental changes by tracking KPIs and using previous hyperparameter values as a warm-up starting point.

Moreover, each individual agent will be required to conduct basic forecasting operations using regularized linear regression, applied to the monitored metrics. To enable autopoietic behavior, the regularization hyperparameter will be self-adjusted using a novel bilevel optimization approach. When feasible, and subject to hardware constraints, an innovative Neural Network architecture named 'split-boost' [28] will also be employed for the same purpose of having autonomous tuning of hyperparameters.

5. Conclusion

This document has presented the foundations of the ACES Knowledge Model, which constitutes the backbone for realizing the autopoietic capabilities of the ACES platform. The pursuit of an autopoietic system, capable of self-management and continuous adaptation, requires the agent components developed in WP4 to be aware of every relevant characteristic of the static and runtime state of the ACES platform, in order to adequately decide on the required course of action. The model's design is a response to the increasing need for sophisticated orchestration of services in dynamic edge-cloud environments, which are characterized by their heterogeneous and decentralized nature.

In the transition "From Data to Knowledge," this deliverable has articulated the transformative process through which raw data is elevated to actionable intelligence within the ACES platform. It presents the blueprint for the architecture that will be involved directly in data collection and telemetry, to be complemented through processing and analytics, to knowledge formation and operational wisdom. This process is facilitated by leveraging advanced data aggregation techniques, robust analytics, and machine learning algorithms that convert the vast streams of telemetry and metrics into a coherent understanding of the system's state and performance.

The Knowledge Model is built upon the NGSI-LD framework. Through the JSON-LD format, we have ensured that the information model is not only standardized but also interoperable across different systems and platforms. The model effectively captures the core aspects of supply and demand within the ACES infrastructure. On the supply side, it details the computational resources, such as CPUs, GPUs, and memory, as well as the network components that constitute the EMDCs. For the demand side, it encapsulates the service requirements, operational SLOs, and deployment strategies, which are crucial for service fulfilment and performance optimization.

The ACES Knowledge Model is complemented with the core agent concepts that underpin its autopoietic behavior. The deliverable has introduced distinct swarm agents—both demand and supply—that operate within a dynamic EMDC environment. These agents will be pivotal to deciding and orchestrating complex actions like pod placement, network optimization, and load balance, whose work will be developed in WP4.

Throughout the document, we have explored the relationships between entities, such as the deployment of services on Kubernetes clusters and the utilization of computational resources by pods. These relationships are integral to the model, as they reflect the real-time state and topology of the system and enable the reasoning agents to perform effective orchestration and management tasks.

In conclusion, the work reported in this deliverable represents a significant step towards realizing a self-sustaining system that is equipped to handle the complexities of modern edge-cloud services. The model is expected to evolve as the platform expands and as new challenges and requirements emerge. Future work in WP 3 will build upon this milestone. T3.1 will focus on refining the model, integrating it with the AI/ML agents to realize the vision of the ACES ecosystem. On the other hand T3.2 will develop the full telemetry infrastructure, managing the captured knowledge,

References

[1] Maturana, H.R., & Varela, F.J. (1980). Autopoiesis and Cognition: The Realization of the Living. D. Reidel Publishing Company.

[2] Briscoe, G., & Dini, P. (2010). Towards autopoietic computing. In Digital Ecosystems: Third International Conference, OPAALS 2010, Revised Selected Papers 3, 199-212.

[3] Corcho, O., Gomez-Perez, A. (2002), A Roadmap to Ontology Specification Languages, 12th International Conference on Knowledge Engineering and Knowledge Management

[4] Lexicon Model for Ontologies: Community Report, 10 May 2016, https://www.w3.org/2016/05/ontolex/

[5] Cooper, L.P., The Power of a Question: A Case Study of Two Organizational Knowledge Capture Systems, 36th Hawaii International Conference on System Sciences (HICSS'03)

[6] Haidrar, S., Bencharqui, H., Anwar, A., Bruel, J. M., & Roudies, O. (2017). REQDL: A requirements description language to support requirements traces generation. In 2017 IEEE 25th International Requirements Engineering Conference Workshops (REW) (pp. 26-35). IEEE.

[7] Hargadon, A.B. (1998), Firms as Knowledge Brokers: Lessons in Pursuing Continuous Innovation, California Management Review

[8] Correa da Silva, F.S. Vasconcelos, W.W., Robertson, D.S. Brilhante, V. de Melo, A.C.V. Finger, M. Agusti, J. (2002), On the insufficiency of ontologies: problems in knowledge sharing and alternative solutions, Knowledge-Based Systems, 15 (3), Pages 147-167

[9] Bencharqui, B. Haidrar. S. Anwar, A. (2022) Ontology-based Requirements Specification Process, E3S Web of Conferences, ICIES'22, 351, 01045 https://doi.org/10.1051/e3sconf/202235101045

[10] Mirsky, Y. et al., Kitsune: An Ensemble of Autoencoders for Online Network Intrusion Detection, NDSS'18

[11] D. Huang, D. Cui,H. Wen, and S. Huang, C. (2019). "Security analysis and threats detection techniques on docker container," in 2019 IEEE 5th International Conference on Computer and Communications (ICCC).

[12] Lin, Y. Tunde-Onadele, O. Gu, X. He, J. and Latapie, H. (2022). "Shil: Self-supervised hybrid learning for security attack detection in containerized applications," in 2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS).

[13] Lin, Y. Tunde-Onadele, O. and Gu, X.(2020). "Cdl: Classified distributed learning for detecting security attacks in containerized applications," in Annual Computer Security Applications Conference, ACSAC '20.

[14] Flora, J. Goncalves, P., and Antunes N.(2020), "Using attack injection to evaluate intrusion detection effectiveness in container-based systems," in 2020 IEEE 25th Pacific Rim International Symposium on Dependable Computing (PRDC).

[15] Nguyen, T., Rieger, P., et al., (2022). FLAME: Taming Backdoors in Federated Learning, In: Proceedings of the 31st USENIX Security Symposium, pp. 1415-1432, USENIX Association, 31st USENIX Security Symposium (USENIX Security 22), Boston, USA, 10.-12.08.2022.

[16] Fereidooni, H. Pegoraro, A., et al. (2024), FreqFed: A Frequency Analysis-Based Approach for Mitigating Poisoning Attacks in Federated Learning, In: Network and Distributed System Security (NDSS) Symposium 2024.

[17] Kumari, K., Rieger P., et al., (2023). BayBFed: Bayesian Backdoor Defense for Federated Learning, In: 44th IEEE Symposium on Security and Privacy (S&P).

[18] Shen, S., Tople, S., and Saxena P., (2016). Auror: Defending Against Poisoning Attacks in Collaborative Deep Learning Systems. In ACSAC.

[19] Wilensky, U., & Rand, W. (2015). An introduction to agent-based modeling: modeling natural, social, and engineered complex systems with NetLogo. Mit Press.

[20] Umlauft, M., Schranz, M., & Elmenreich, W. (2023). Simulation of Swarm Intelligence for Flexible Job-Shop Scheduling with SwarmFabSim: Case Studies with Artificial Hormones and an Ant Algorithm. Springer Book Chapter (LNNS).
[21] Schranz, M., Umlauft, M., & Elmenreich, W. (2021). Bottom-up Job Shop Scheduling with Swarm Intelligence in Large Production Plants. In SIMULTECH (pp. 327-334).
[22] Schranz, M., Harshina, K., Forgacs, P., & Buining, F. (2024). Agent-based Modeling in the Edge Continuum using Swarm Intelligence. In ICAART (under review).
[23] Gad, A. (2022). Particle swarm optimization algorithm and its applications: a systematic review. Archives of computational methods in engineering, 29.5.
[24] Ali, Y. A., Awwad, E. M., Al-Razgan, M., & Maarouf, A. (2023). Hyperparameter search for machine learning algorithms for optimizing the computational complexity. Processes, 11(2), 349.
[25] Bemporad, A. (2020). Global optimization via inverse distance weighting and radial basis functions. Computational Optimization and Applications, 77(2), 571-595.
[26] Sutton, R. S., & Barto, A. G. (1999). Reinforcement learning: An introduction. Robotica, 17(2), 229-235.
[27] Cannelli, L., Zhu, M., Farina, F., Bemporad, A., & Piga, D. (2023). Multi-agent active learning for distributed black-box optimization. IEEE Control Systems Letters.
[28] Cestari, R. G., Maroni, G., Cannelli, L., Piga, D., & Formentin, S. (2023). Split-Boost Neural Networks. arXiv preprint arXiv:2309.03167.

© 2023