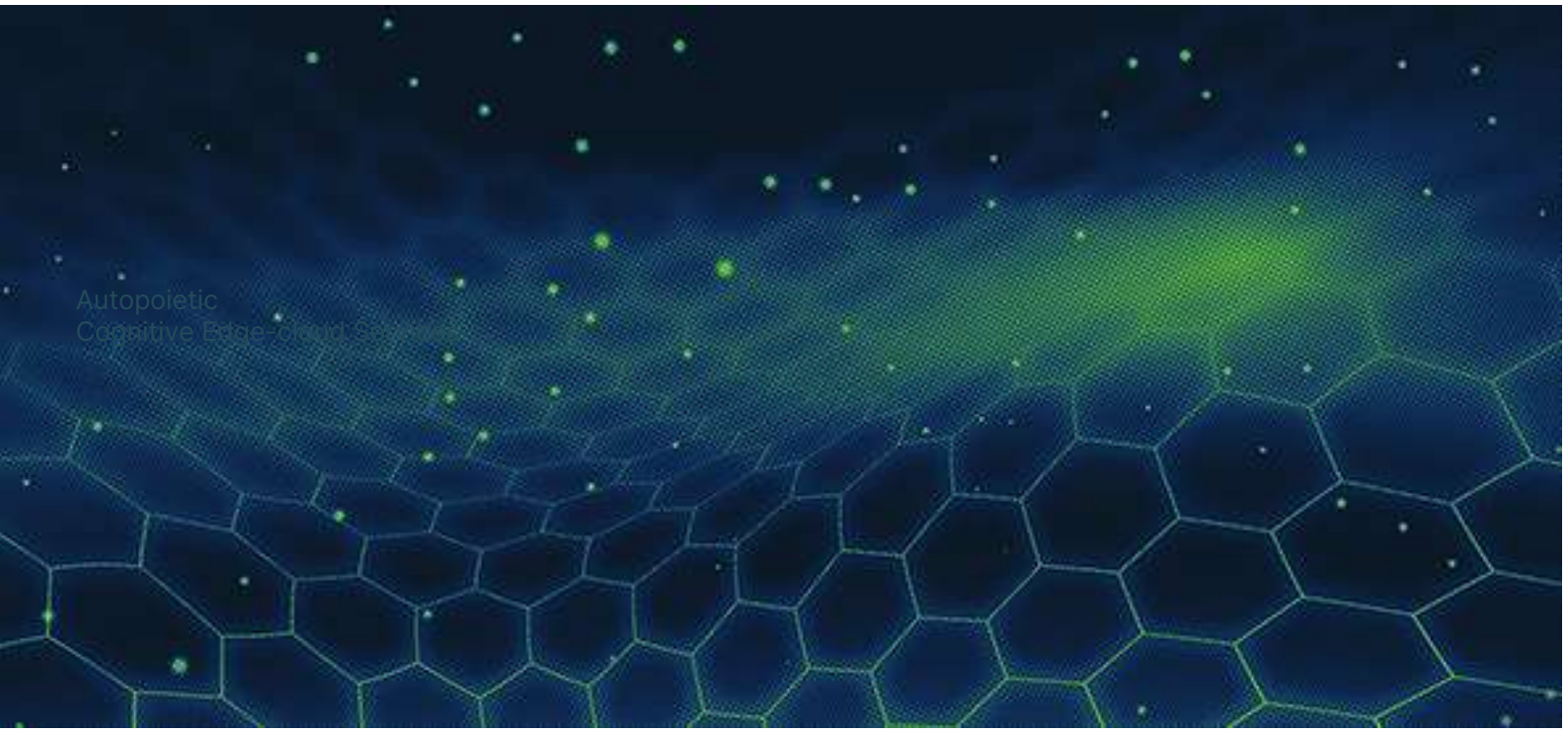

Deliverable 4.5
Distributed knowledge base and data management systems

Grant Agreement Number: 101093126

hiro MICRODATACENTERS

inesc id lisboa

ipto

sixsq

Lakeside Labs SELF-ORGANIZING NETWORKED SYSTEMS

POLITÉCNICA

MARTEL Innovate

TECHNISCHE UNIVERSITÄT DARMSTADT

University of Ljubljana Faculty of Computer and Information Science

IDSIA SUPSI

DATAPOWER CONSULTING

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).

D4.5 – Distributed knowledge base and data management systems	
Author(s)	Loris Cannelli, Vito Cianchini, Félix Cuadrado, Rui Min, Melanie Schranz, Konstantin Skaburskas, Aymen Yahyaoui
Editor	Loris Cannelli
Participating partners	HIRO, LAKESIDE, MARTEL, SIXSQ, SUPSI, UPM
Version	Final
Status	Completed
Deliverable date	M30
Dissemination Lvl	PU - Public
Official date	30 June 2025
Actual date	1 July 2025

Executive Summary

The ACES (Autopoietic Cognitive Edge-cloud Services) project aims to develop a next-generation intelligent edge-cloud continuum, capable of self-management, adaptation, and autonomous decision-making in complex, distributed environments. Within this broader vision, **Deliverable D4.5** addresses one of the core technological pillars of ACES: the creation of **distributed knowledge base and data management systems** that can operate reliably, coherently, and adaptively across a highly heterogeneous and dynamic infrastructure.

This deliverable explores how knowledge can be structured, distributed, and leveraged at the edge to support **cognitive behaviour**, such as local reasoning, anomaly detection, and self-optimization. Moving beyond traditional cloud paradigms, ACES promotes a decentralized, data-centric approach, where edge nodes are not merely consumers of instructions but intelligent agents that observe, reason, and act locally—while maintaining alignment with global goals.

A key innovation presented in this document is the integration of **Temporal Knowledge Graphs** and semantic models, which allow services to represent evolving system states and causal dependencies over time. These representations underpin anomaly detection, root cause analysis, and service coordination, enabling the system to reason about time-sensitive relationships and to anticipate failure propagation in microservice-based environments. Machine learning and graph-based techniques are used in tandem to infer and interpret complex system dynamics in a scalable and explainable way.

To orchestrate resources and services efficiently across the edge-cloud landscape, ACES introduces **multi-agent models and decentralized optimization techniques**. Swarm intelligence and learning-based peer selection mechanisms are employed to allocate workloads based on local observations and partial knowledge, while preserving global performance and service-level guarantees. In this context, the deliverable demonstrates how **local surrogate models and inverse-distance exploration strategies** can be employed to tune system behaviour dynamically, without central coordination.

Another foundational component of this deliverable is the **distributed data management infrastructure**. Rather than relying on centralized repositories, ACES deploys a hybrid architecture where telemetry, metadata, and control flows are processed locally, while global coordination is achieved through lightweight metadata exchange and policy-driven decisions. By embedding intelligence directly into the data layer—via metadata tags, policies, and cognitive agents—the system can respond autonomously to changes in load, failures, or network conditions, achieving a high degree of **resilience and adaptivity**.

In summary, the Deliverable presents a coherent and integrated vision for enabling **autopoietic intelligence** in edge-cloud networks. It contributes essential architectural, algorithmic, and operational mechanisms that transform the edge from a passive layer into an active participant in the global system's intelligence. These contributions are central to ACES's ambition to realize distributed, explainable, and self-sustaining cognitive systems that can operate reliably under uncertainty, at scale, and in real-world industrial and societal contexts.

Disclaimer

The ACES consortium consists of the following partners:

No	PARTNER ORGANISATION NAME	ABBREVIATION	COUNTRY
1	INSTITUTO DE ENGENHARIA DE SISTEMAS E COMPUTADORES, INVESTIGACAO E DESENVOLVIMENTO EM LISBOA	INESC ID	PT
2	HIRO MICRODATACENTERS B.V	HIRO	NL
3	TECHNISCHE UNIVERSITAT DARMSTADT	TUD	DE
4	LAKESIDE LABS GMBH	LAKE	AT
5	UNIVERZA V LJUBLJANI	UL	SI
6	UNIVERSIDAD POLITECNICA DE MADRID	UPM	ES
7	MARTEL GMBH	MAR	CH
8	SCUOLA UNIVERSITARIA PROFESSIONALE DELLA SVIZZERA ITALIANA	SUPSI	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
07/03/2025	1.0	Table of Contents	SUPSI
01/04/2025	1.1	Revised Table of Contents	HIRO, LAKESIDE, MARTEL, SIXSQ, SUPSI, UPM
15/04/2025	1.2	Finalized Table of Contents	SUPSI
20/05/2025	1.3	Contributions	HIRO, LAKESIDE, MARTEL, SIXSQ, SUPSI, UPM
28/05/2025	2.0	Ready-for-review version	SUPSI
23/06/2025	2.1	Review completed	HIRO
01/07/2025	2.2	Final version after project coordination review	INESC

Authors

AUTHOR	PARTNER
Loris Cannelli	SUPSI
Félix Cuadrado, Aymen Yahyaoui	UPM
Vito Cianchini	MARTEL
Konstantin Skaburskas	SIXSQ
Melanie Schranz	LAKESIDE
Rui Min	HIRO

Reviewers

NAME	ORGANISATION
Fernando Ramos	INESC ID
Maksimilian Mamiliaev	HIRO

List of terms and abbreviations

ABBREVIATION	DESCRIPTION
ABM	Agent-Based Modelling
API	Application Programming Interface
BO	Bayesian Optimization
CF	Cognitive Framework
CPU	Central Processing Unit
CRDT	Conflict-free Replicated Data Type
CXL	Compute eXpress Link
DKB	Distributed Knowledge Base
DKBRS	Distributed Knowledge Base Replication Service
EMDC	Edge Micro Data Center
ETL	Extract Transform Load
GAE	Graph Auto Encoder
GCN	Graph Convolutional Network
GNN	Graph Neural Network
GPU	Graphics Processing Unit
gRPC	Google Remote Procedure Calls
HTTP	Hyper Text Transfer Protocol
IDW	Inverse Distance Weighting
IoT	Internet of Things
I/O	Input/Output

KG	Knowledge Graph
KNN	K Nearest Neighbour
KPI	Key Performance Indicator
LC	Latency Critical
LRA	Long-Running Application
ML	Machine Learning
MLP	Multi Layer Perceptron
MSE	Mean Squared Error
NATS	Neural Autonomic Transport System
NN	Neural Network
PCIe	Peripheral Component Interconnect Express
QoS	Quality of Service
RAM	Random Access Memory
RBF	Radial Basis Function
ReLU	Rectified Linear Unit
SLA	Service Level Agreement
SVM	Support Vector Machine
SQL	Structured Query Language
TKG	Temporal Knowledge Graph

Table of contents

1 Introduction

The present document focuses on the development of distributed knowledge base and data management systems within ACES. It consolidates the methodologies, technologies, and experiments that enable the system to observe, reason, and adapt to evolving conditions, while maintaining data locality, low-latency operations, and coordination across heterogeneous infrastructures.

The document is structured as follows:

- Section 2 outlines the key challenges in distributed knowledge and data management, such as data heterogeneity, semantic interoperability, scalability, and temporal reasoning, and introduces ACES strategies to address them.
- Section 3 presents the design and implementation of Temporal Knowledge Graphs (TKGs) for anomaly detection and root cause analysis in microservice-based architectures, including machine learning and graph-based methods.
- Section 4 describes a self-organizing local decision mechanism based on agent-based modelling and swarm intelligence, addressing adaptive scheduling and resource orchestration in edge environments.
- Section 5 details a multi-agent AI algorithm for decentralized resource allocation using the D-GLIS framework, which enables distributed hyperparameter tuning and system optimization.
- Section 6 discusses the distributed data management layer, including operational principles, metadata-driven decisions, and integration with telemetry systems to support autonomous edge-cloud behaviours.
- Section 7 provides the concluding remarks, summarizing the contributions and highlighting future directions.

Together, these sections aim to demonstrate how ACES leverages cognitive frameworks, distributed intelligence, and advanced data handling to enable the emergence of self-managing, resilient edge-cloud systems.

2 Challenges in Distributed Knowledge and Data Management

In the agent-based modelling of an EMDC (see D4.1 on agent-based modelling for details), we face a set of challenges that need to be considered in the overall modelling process.

2.1 Pool of Resources

To begin with, together with 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 CXL is being implemented in CPUs (see, e.g., Intel, or AMD), in memory and storage (e.g., Samsung), and the PCIe switches are expected in 2026. Besides hardware development, the biggest challenge currently is how these pools of resources can be orchestrated, as there is no scheduling technique available to handle this complexity and dynamics in binding requested resources.

2.2 Application Types

For the different services, we can differ between the three application types that come with diverse requirements in their response time:

1. The LRAs instantiate long-standing pods to enable iterative computations in memory or unceasing request-response. LRAs include processing frameworks (e.g., Storm, Flink, Kafka streams), latency-sensitive database applications (e.g., HBase and MongoDB), and data-intensive in-memory computing frameworks (e.g., TensorFlow).
2. 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.
3. Stream processing also deals with large volumes of data, but the data needs to be processed in real time. 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.

2.3 Relationships among Pods

The demand swarm agents are related pod splits from a specific service. 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 service s. Currently, these relationships are not used in the scheduler and orchestration optimization. For example, placing interacting services closer together can significantly enhance their performance, e.g.: i) 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; ii) 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.

2.4 Additional Challenges and ACES Strategies for Distributed Knowledge and Data Management

The ACES project approaches distributed knowledge and data management with a data-centric and cognitive-by-design methodology. Below is a breakdown of critical challenges and the mechanisms adopted in ACES to address them.

2.4.1 Data Heterogeneity

In distributed environments, data originates from diverse sources: IoT sensors, cloud nodes, edge services, often using incompatible formats, schemas, and models.
ACES employs a graph-based data model aligned with NGSI-LD [1] to provide semantic structure across all telemetry data. Data from various storage systems: TimescaleDB (for time-series), Neo4j (for graphs), MinIO (for objects) is transformed into this unified semantic format, enabling interoperability and seamless integration across services and nodes.

2.4.2 Semantic Interoperability

Systems must align not only data formats but also the meaning behind data (e.g., context of metrics, alerts, or relationships).
By using semantic enrichment via its NGSI-LD-compatible data model, ACES supports machine-readable ontologies and context-aware relationships. NATS-based pipelines [2] transform raw telemetry into enriched knowledge, which agents then consume through the CF for inference, correlation, and action planning.

2.4.3 Scalability

Managing the growth of data sources, query load, and event streams without overwhelming the system.
The architecture is modular and horizontally scalable, with NATS enabling scalable and decoupled ingestion of telemetry. Agents are distributed and operate locally, consuming only relevant data, which reduces central bottlenecks. Retention policies and ETL pipelines orchestrated by Prefect offload historical data to MinIO, ensuring scalability of live systems.

2.4.4 Knowledge representation and reasoning

Storing data is not enough, systems must interpret and reason over it to enable autonomous behaviour. The Cognitive Framework in ACES integrates telemetry, metadata, and graph-based knowledge representations. This allows for machine learning-based reasoning and inference of system states, such as detecting anomalies or predicting failures. Data is not passive since it's actively interpreted to influence orchestration decisions in real time.

2.4.5 Data Governance and Ownership

In collaborative and multi-stakeholder environments, it is critical to manage data provenance, ownership, and lifecycle transparently.
ACES uses a metadata-rich governance model. Every data item includes tags for origin, type, usage context, and versioning, enabling cognitive agents to enforce retention policies, trigger ETL workflows via Prefect, and track data lineage across services. The separation of payloads and metadata supports global coordination without compromising edge autonomy.

2.4.6 Temporal Knowledge graph representation

While a knowledge graph (KG), which we explained in D3.3, concentrates on static relationships and entities without explicit time tracking, a temporal knowledge graph (TKG) captures and portrays data with a time dimension, demonstrating how entities and relationships change over time. TKGs are one of the approaches used in ACES to model microservice interactions and resource usage for anomaly

detection and root cause analysis; however, they face several significant challenges as well. First, integrating many data sources, including service traces and resource usage metrics and logs, into a single temporal graph representation is more difficult due to the microservice environments' dynamic and heterogeneous nature. Second, autoscaling and frequent deployments cause the topology to change quickly, which creates difficult problems for graph generation and updates. Furthermore, to guarantee significant temporal reasoning, temporal granularity and synchronization need to be well-controlled. When it comes to learning, GNNs frequently face challenges with generalization across system versions, scalability, and label availability. Effective root cause analysis still faces significant challenges in separating causal linkages from correlations and guaranteeing explainability. Lastly, thorough assessment and benchmarking of suggested approaches are severely constrained by the absence of consistent datasets and ground truth annotations.

3 Temporal Knowledge graphs for anomaly detection and root cause analysis

3.1 Temporal Knowledge Graphs for ACES

A KG is an organized depiction of information regarding entities and their relationships. These graphs are inherently static, indicating they presume facts are perpetually accurate and do not consider when these facts hold validity. Conversely, a TKG enhances this model by adding temporal details, doting every fact with a timestamp. This enables TKGs to reflect the dynamic progression of knowledge over time, making them vital for applications where the timing and order of events are crucial, such as event prediction, anomaly identification, and root cause exploration in complex systems like microservices. While traditional KGs are good at representing consistent global information, TKGs are better at analysing historical contexts, temporal trends, and predicting future events based on past behaviour.

In recent years, Kubernetes has become the top orchestration technology for containerized microservice application deployment and management. These applications are defined by their distributed characteristics, where microservices interact with one another across networks, frequently exhibiting different degrees of resource usage. Kubernetes [3] provides robust tools for automating the deployment and scaling of microservices, but it also brings challenges related to reliability and fault tolerance. Failures in microservice applications can stem from multiple sources, including resource depletion, network disruptions, or application failures. If not managed effectively, these failures can diminish service quality and result in considerable operational costs. For instance, when pods are overloaded or the network is congested, there may be an increase in latency between microservices, which can cause slower response times and delayed communication. Service continuity is disrupted, and overall application performance is degraded by frequent pod restarts and resource depletion, such as CPU throttling or memory exhaustion. Anticipating failures in these dynamic settings is a challenging task, as the relationships among microservices and their resource consumption are always evolving. A promising method for modeling these systems involves using knowledge graphs, especially TKGs, which represent evolving relationships among entities over time. In the realm of Kubernetes, TKGs can signify the interaction between pods, their utilization of resources, and their generated events. Examining these graphs could potentially reveal patterns or irregularities that occur before failures.

In the context of the ACES project, we consider a distributed system that continuously monitors its own performance. A TKG can represent internal service states, interdependencies, configuration changes, and observed anomalies over time. When a service degrades, the system can query the TKG to infer probable causes and trigger self-healing actions, effectively making the system observe, reason, and reorganize itself, which leads to the intended autopoietic behaviour in this project.

3.2Microservice application deployment

To evaluate anomaly detection methods in a realistic cloud-native environment, we utilized the Online Boutique application, which is an open-source microservice-based e-commerce platform [4] developed by Google as a reference workload for Kubernetes environments. This application serves as a representative benchmark for modern distributed systems, exhibiting many characteristics typical of production microservices architectures, such as service decomposition, inter-service communication, and dynamic scaling.

The Online Boutique consists of over ten loosely coupled services, as shown in Figure 1, which is an example of TKG showing the different microservices of this application. Each microservice is responsible for a specific business function, such as product catalog management, cart handling, payment processing, shipping, currency conversion, recommendation generation, and user authentication. These microservices are usually installed in containers inside a Kubernetes cluster and communicate using the HTTP and gRPC protocols. Service discovery, load balancing, and observability through telemetry data such as logs, metrics, and traces are examples of real-world cloud-native techniques that are reflected in the architecture.

This application provides a rich testbed for experimenting with anomaly injection and detection. Its modular design enables targeted injection of faults such as CPU saturation, memory leaks, network latency, or packet loss at the level of individual services. In addition, the availability of performance metrics and service-to-service communication data allows for detailed monitoring and root cause analysis.

By leveraging the Online Boutique, we ensure that our evaluation is grounded in a realistic operational setting, offering a practical context for testing the effectiveness of machine learning-based anomaly detection techniques in cloud-native environments.

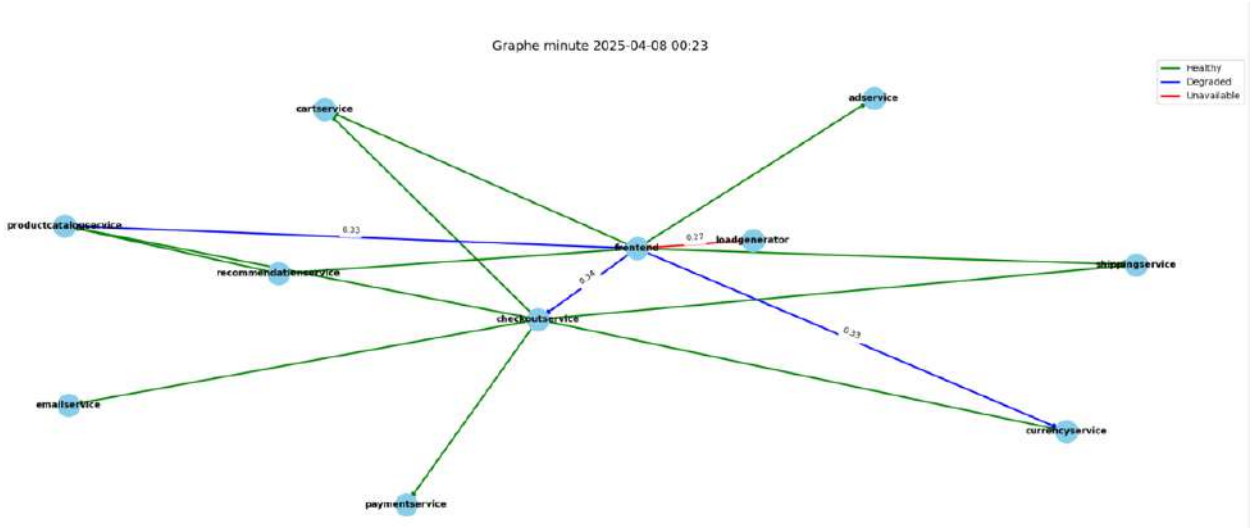

Figure 1: Online boutique application example of TKG

3.3Faults Injection

An automated framework was developed to streamline application testing and fault injection.

This framework is a specialized service designed to simulate and automate diverse failure scenarios within a controlled environment. It offers the possibility to inject targeted faults as described in Table 1 such as CPU hog, memory leak, network latency/loss, http delays and pods crashes while allowing configuration of parameters like duration, intensity, and affected services. The component logs detailed information about each injected failure to facilitate correlation with observed system behaviour. Seamlessly integrating with Kubernetes and Istio APIs [5], it can impose failures at various levels of the stack. Its implementation harnesses Kubernetes resource controls and Istio's traffic management to emulate realistic failure conditions without compromising overall system stability.

Fault Type	Effect	Tools / Cmds	Typical Usage
CPU hog	Overloads CPU, causing performance degradation	stress-ng	Test system performance under heavy computation
Memory leak	Simulates memory exhaustion and potential crashes	stress-ng	Test system stability under high memory usage
Network loss	Simulates packet loss in the network, causing connectivity issues	tc (traffic control)	Test system resilience to network issues
Network Delay	Introduces latency in network communication, simulating slow network conditions	tc (traffic control)	Test system behaviour under network latency
HTTP delay	Simulates slow HTTP responses (like a failing backend)	Istio	Test application resilience (e.g., slow HTTP responses).
Pod crash/unavailability	Introducing faults that cause pods to crash (e.g., an unhandled exception triggered by specific input, extreme load)	stress-ng/ kubectl delete	Tests service availability

Table 1: Injected failure types

This framework typically performs the following functions:

- Interact with the Kubernetes cluster (via the Python Kubernetes client library) to identify target pods for fault injection.
- Execute commands within the target pods using 'kubectl exec' to trigger 'stress-ng' or 'tc' tools.
  For example, to inject CPU stress into a pod:
  kubectl exec <pod-name> -n <namespace> -- stress-ng --cpu 1 --cpu-load 80 –timeout 60s
  To introduce network latency to a pod's network interface (e.g., 'eth0'):
  kubectl exec <pod-name> -n <namespace> -- tc qdisc add dev eth0 root netem delay 100ms
- Monitor the application's behaviour during and after the fault injection, possibly by querying Prometheus or observing application logs.
- Collect data and construct labelled datasets to be used for ML models.

We show in Figure 2 an example of a fault injection scenario, where all parameters were specified, such as the namespace, the number of users to interact with the application simulating loads, failure type with its options, and durations of the experiment and the failure.

```
[?] Namespace: ob
[?] Number of users (multipe of 100): 3
[?] Load duration (minutes): 6
[INPUT] Choose failure type:
1) CPU stress
2) Memory stress
3) Network loss (tc)
4) Network delay (tc)
5) Delay injection (Istio)
[?] Enter option number (1-5): 1
[?] Failure duration (seconds, default 300): 180
[?] Time to wait before failure injection (seconds, default 180): 180
[?] CPU workers (default 2):
[?] Target microservices (space-separated, except for email service and recommendation service which use stress and not stress-ng): adservice
[INFO] Scaling loadgenerator...
deployment.apps/loadgenerator scaled
[INFO] Waiting 180 seconds before injecting failure...
[INFO] Injecting CPU into adservice pod: adservice-5dc4c759b6-b9dwc
```

Figure 2: Fault injection scenario example

We show in Figure 3 and Figure 4 below examples of failure results. We notice a change in CPU utilization from **35m** to **213m** at **21:11:35** after injecting a CPU failure on the microservice Adservice.

```
2025-04-08 21:05:33,adservice-7d94c4d5dd-hm24b,36m,183Mi
2025-04-08 21:06:34,adservice-7d94c4d5dd-hm24b,37m,183Mi
2025-04-08 21:07:34,adservice-7d94c4d5dd-hm24b,35m,184Mi
2025-04-08 21:11:35,adservice-7d94c4d5dd-hm24b,213m,184Mi
2025-04-08 21:12:35,adservice-7d94c4d5dd-hm24b,37m,183Mi
2025-04-08 21:13:36,adservice-7d94c4d5dd-hm24b,35m,184Mi
```

Figure 3: Metric results after CPU Fault Injection on Adservice

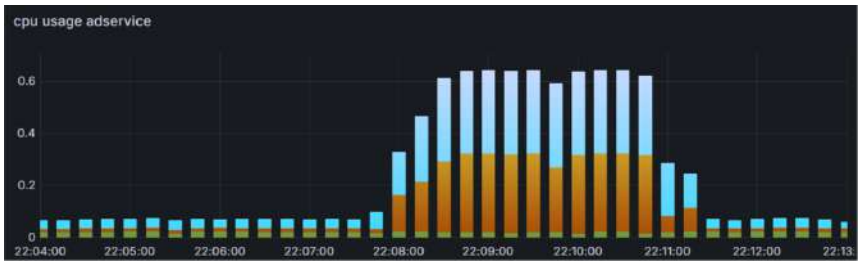

Figure 4: Result graph of CPU Fault Injection on Adservice

3.4 Data Collection and aggregation

To support the detection and classification of anomalies in microservice architectures, we collect two types of data: resource consumption and microservice communications.

3.4.1 Resource consumption data

After performing several experiments (more than 100 experiments: 4 fault types x 5 target MS x 4 repeat times changing parameters) of fault injection and creating datasets of normal and abnormal data of different types, we automate structuring and annotation of experimental data. This process enables automated preparation of datasets across multiple failure scenarios and ensures consistency throughout the processing pipeline. The key steps of this automated procedure are as follows:

- **Automated Traversal of Experiment Subdirectories and Failure Type Identification**: The process systematically navigates through experimental subfolders and infers the failure type based on filename suffixes (e.g., _cpu, _mem, _loss, _delay), corresponding respectively to CPU hog, memory leak, packet loss, and packet delay scenarios.

- **Data Ingestion and Structuring**: It automatically loads the raw resource and communication logs for each pod, originally stored in dedicated files, and applies cleaning and formatting

operations to produce structured datasets suitable for analysis.

- **Temporal Annotation of Abnormality Classes**: Each data entry is labelled either as *Normal* or assigned a specific anomaly class. This labelling is based on a fixed injection timeline, where anomalies are assumed to begin three minutes after the start of each experiment.

- **Merging of Experiment Datasets**: Data from all individual experiments are consolidated into a unified dataset, enabling holistic analysis across all failure types.

- **Chronological Sorting and Instance Name Standardization**: To ensure temporal consistency and simplify further processing, all records are chronologically ordered, and instance names are standardized.
- **Final Export for Downstream Tasks**: The structured and annotated datasets are exported for subsequent use in model training, evaluation, and visualization, including resource metrics and inter-pod communication data.

This preprocessing pipeline provides a scalable and reproducible foundation for the study of anomaly detection in microservice environments, enabling the use of labelled, temporally aligned datasets across a variety of failure scenarios.

3.4.2 Communication data

The communication between microservices was analysed to differentiate between successful and failed interactions. This distinction was made using HTTP and gRPC response codes, allowing for a clearer understanding of system behaviour under both normal and faulty conditions. The data processing workflow involved the following key steps:

- **Classification of Communication Outcomes**: Microservice interactions were labelled as either *success* or *error* based on standard response codes, with HTTP 200 and corresponding gRPC statuses used to identify successful communications.

- **Feature Engineering**: Several derived metrics were computed to capture dynamic system behaviour, including the number of new requests, the volume of data exchanged, and communication latency calculated as the time difference between successive requests.

- **Aggregation of Temporal Metrics**: After classification and metric derivation, the data was aggregated to extract KPIs over time. These indicators are essential for evaluating the reliability and efficiency of microservice-to-microservice interactions.

The computed KPIs include:

- **Success and Error Rates:** reflecting the proportion of failed versus successful communications.
- **Average latency:** measuring the overall responsiveness of the system across all requests.
- **Throughput:** representing the amount of data transmitted per unit time.
- **Request rate:** indicating the number of microservice calls processed per second.

These metrics were calculated across multiple time resolutions (e.g., 15 seconds, 30 seconds, 1 minute) to support detailed performance monitoring over short-, medium-, and long-term intervals. The resulting dataset serves as a foundation for training and evaluating machine learning models in anomaly detection.

3.5ML-based Anomaly Detection

We aim to detect and classify the various types of anomalies stated in section 3.2 occurring within microservices by leveraging traditional machine learning techniques. Microservice architectures, while offering scalability and flexibility, introduce complexity that makes anomaly detection a critical challenge. To address this, we extract relevant features related to resource consumption, such as

- **cpu_usage_seconds_total:** Measures the total cumulative CPU time consumed by a container in seconds
- **container_cpu_system_seconds_total:** Indicates the total time the container has spent executing system (kernel) level operations
- **container_memory_working_set_bytes:** Represents the amount of memory in active use by the container (excluding cached pages that can be evicted)
- **container_memory_rss:** Shows the portion of memory occupied by a container that cannot be swapped out (resident set size); useful for understanding real physical memory usage.
- **container_network_receive_bytes_total:** Tracks the total number of bytes received over the network by a container; helps monitor incoming traffic volume and potential bottlenecks.
- **container_network_transmit_packets_total:** Counts the total number of packets sent over the network by the container; indicates outbound communication frequency and traffic patterns.

We apply a series of supervised learning models to not only identify the presence of anomalies but also predict their specific types. This approach provides a foundation for proactive monitoring and automated root cause analysis in distributed environment. To do that, we proceed by doing the following steps:

- **Data Preprocessing**: Encoding of categorical variables and imputation of missing values.
- **Normalization**: StandardScaler is used to ensure balance among variables.
- **Dataset Splitting**: 80% for training and 20% for testing, with stratification based on classes.
- **Tested Models**: Random Forest, Decision Tree, SVM, KNN, and Naive Bayes.

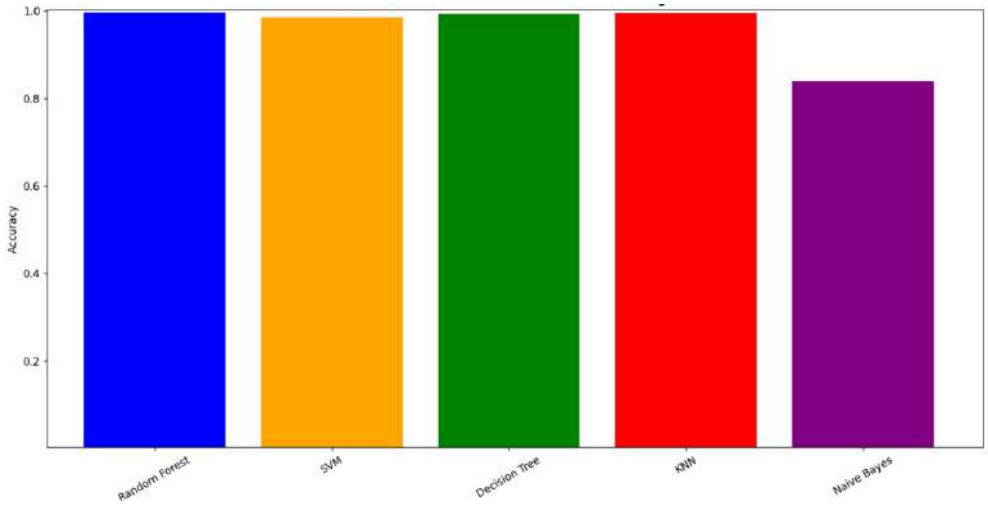

Figure 5: Results of classification of the Anomalies using ML algorithms

Results shown in Figure 5 above depict high accuracy for detecting anomalies (over 95%) for most of the used algorithms except naive Bayes. Random Forest gave the highest accuracy; therefore, it was chosen for the detection task.

3.6 GNN-based communication anomaly detection

To detect anomalies in microservice communications, a Graph Autoencoder (GAE) model was employed. This model learns a latent representation of the communication graph, allowing it to identify abnormal interactions between services. The architecture includes:

- A **GCN encoder** that captures both node features (e.g., latency, error rate) and graph structure.
- A **decoder** that reconstructs graph edges from the encoded node embeddings.

The model is trained using **binary cross-entropy loss** to reconstruct edges accurately. Anomalies are detected based on **low reconstruction probabilities**—i.e., edges that the model fails to predict well are flagged as abnormal.

- **Anomaly scores** are computed as 1 - reconstruction probability, with higher scores indicating more suspicious behaviour.
- **Training parameters:** Optimizer: Adam, Learning rate: 0.0005, Epochs: 200.

The resulting anomaly scores can be used for **alerting**, **dashboard integration**, or **graph-based system visualization**, providing a powerful unsupervised method for detecting communication failures in microservices.

3.7 Temporal Knowledge graphs

To enhance interpretability and support root cause analysis in microservice architectures, we leverage **TKG visualizations** that integrate traces of millisecond communication and resources metrics insights from anomaly detection models over time. The knowledge graph evolves as the system operates, representing both microservices and workers nodes (as nodes) and microservices communications and their deployment on worker nodes (as directed edges), enriched with temporal and behavioural annotations. Two detection layers are visualized concurrently:

- **Edge-level anomalies**, derived from a **GAE**, identify abnormal service interactions based on deviations in graph structure.
- **Node-level states**, predicted by a **Random Forest classifier**, categorize each microservice's operational condition (e.g., CPU overload, memory leak).

3.7.1 Visual Encoding and Metrics

The TKG is enhanced with color-coded cues:

- **Edges**: Green (normal), Blue (degraded), Red (anomalous) based on GAE anomaly scores.
- **Nodes**: Coloured or shaped based on the fault class or healthy status from the classifier.

Each node is further enriched with real-time metrics (CPU, memory, network), linking model outputs to observable system behaviour.

3.7.2 Temporal Analysis

An interactive timeline facilitates **dynamic exploration** of the system's state across time, enabling:

- Tracking of anomaly onset and propagation,
- Temporal correlation between service failures and degraded communications,
- Analysis of fault persistence or resolution.

This TKG-based visualization, as shown in Figure 6, provides a comprehensive, time-aware perspective on microservice health, combining graph-structured anomaly detection with supervised fault classification to support informed monitoring and diagnosis.

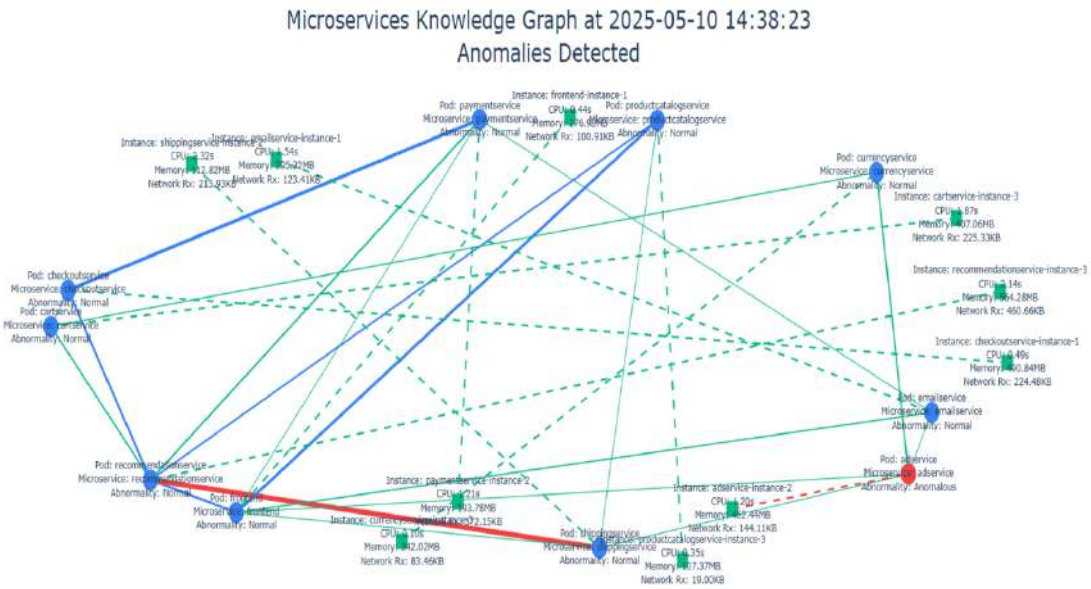

Figure 6: Example of TKG visualization

4 Self-Organizing Local Decision Mechanism

The work carried out in ACES introduces self-organizing local decision mechanisms to address the inherent uncertainty in resource management for mixed workload types and heterogeneous pod requirements. We model edge infrastructure as a multi-agent system comprising:

- Rigid pods with strict CPU/RAM requirements and fixed execution windows
- Elastic pods exploiting residual resources through flexible scheduling
- Worker nodes managing localized resource allocation

Our novel contribution lies in a swarm intelligence-inspired algorithm that:

1. Maintains coarse-grained resource footprints in temporal buckets
2. Implements probabilistic selection of compatible rigid pods
3. Enables decentralized coordination through uniform sampling
4. Evaluates NN-based approaches for deployed pod characterization

4.1 Related Work to the Self-Organizing Approach

Containers represent portable, lightweight units that encapsulate an application and all its dependencies, enabling deployment across diverse environments. Containerization technology has become fundamental in cloud-native architectures, offering flexibility for continuous integration and continuous deployment, regardless of the underlying infrastructure. This approach also enhances resource management in cloud environments, as containers require more fine-grained resource allocation compared to earlier technologies based on virtual machines. Container orchestration frameworks such as Kubernetes automate the deployment and management of containers at scale [6]. Within Kubernetes, which is among the most widely adopted orchestration frameworks for edge computing, the pod serves as the smallest deployable unit and can host one or more containers. At the core of Kubernetes lies its scheduling algorithm, responsible for assigning incoming pods to edge nodes based on required resources (such as CPU and RAM), QoS needs like response time, and system objectives including load balancing, resource utilization, and reliability. Intelligent scheduling algorithms are crucial for improving resource utilization within each cluster [7]. However, the default rule-based scheduling mechanism often falls short in assigning resources efficiently. When the scheduler relies on user-specified resource requirements, significant resource waste can occur, as applications tend to overestimate their needs. The conservative scheduler allocates resources according to each pod's declared demand to ensure QoS, but empirical studies in both Google™ and private clouds have revealed substantial CPU and RAM slack resources [8], [9]. This waste arises from improper scheduling-either by assigning tasks to unsuitable workers or when tasks underutilize their allocated resources.

To address this, schedulers must consider application QoS requirements, typically reflected in resource demand. Analysis of Alibaba™ cloud server traces in [10] highlights that schedulers handle two main classes of applications: LC and batch processing. Co-locating these application types introduces additional flexibility to enhance resource utilization.

Resource waste due to Kubernetes scheduling becomes even more pronounced in edge computing, where environments are more dynamic and system loads can be heavy [11]. Edge clusters often handle a greater proportion of LC workloads and operate with more limited resources compared to the cloud. Given these challenges, resource utilization can be improved by more accurately estimating pod resource demands and worker availability, as well as through strategic resource oversubscription. The former approach leverages advanced machine learning tools to analyse historical utilization data,

enabling more precise resource limits than those set manually by users. Additionally, inferring pod arrival patterns-such as periodic request trends-can further optimize scheduling decisions [12]. The latter approach involves deliberately oversubscribing pods beyond worker capacity to maximize resource use [13]. However, the main challenge here is the careful co-location of LC and batch processing applications, as improper oversubscription can negatively impact the QoS for latency-critical workloads [10].

Both strategies face difficulties in highly dynamic edge environments, where accurately predicting workload demands and meeting application delay requirements is increasingly complex. In this paper, we propose a bottom-up resource allocation strategy: first, allocate resources to strict-demand (rigid) pods, then exploit any slack resources to serve elastic pods that can tolerate greater delays.
In the Kubernetes ecosystem, this method aligns with vertical autoscaling, which complements the more common horizontal autoscaling feature. While horizontal autoscaling adjusts the number of pod replicas to respond to changing workloads, vertical autoscaling fine-tunes the resource allocation for each individual pod [8].

4.2System Model

We consider a general architecture for a distributed EMDC, composed of multiple independently managed clusters (see Figure 7). Each cluster is overseen by a cluster master agent, responsible for managing resource allocation and utilization within the cluster, while also cooperating with other clusters in a self-organizing fashion. Pods are submitted to the cluster master agent based on factors such as location and user preferences and are placed in the master's queue. The master agent can either assign a pod to a worker agent within the same cluster or, if necessary and permitted by the pod's attributes, transfer it to a peer cluster. Peer clusters are trusted, geographically proximate, and accessible via low-latency communication links. In this work, we focus exclusively on intra-cluster resource assignment for pods, and do not address inter-cluster orchestration.

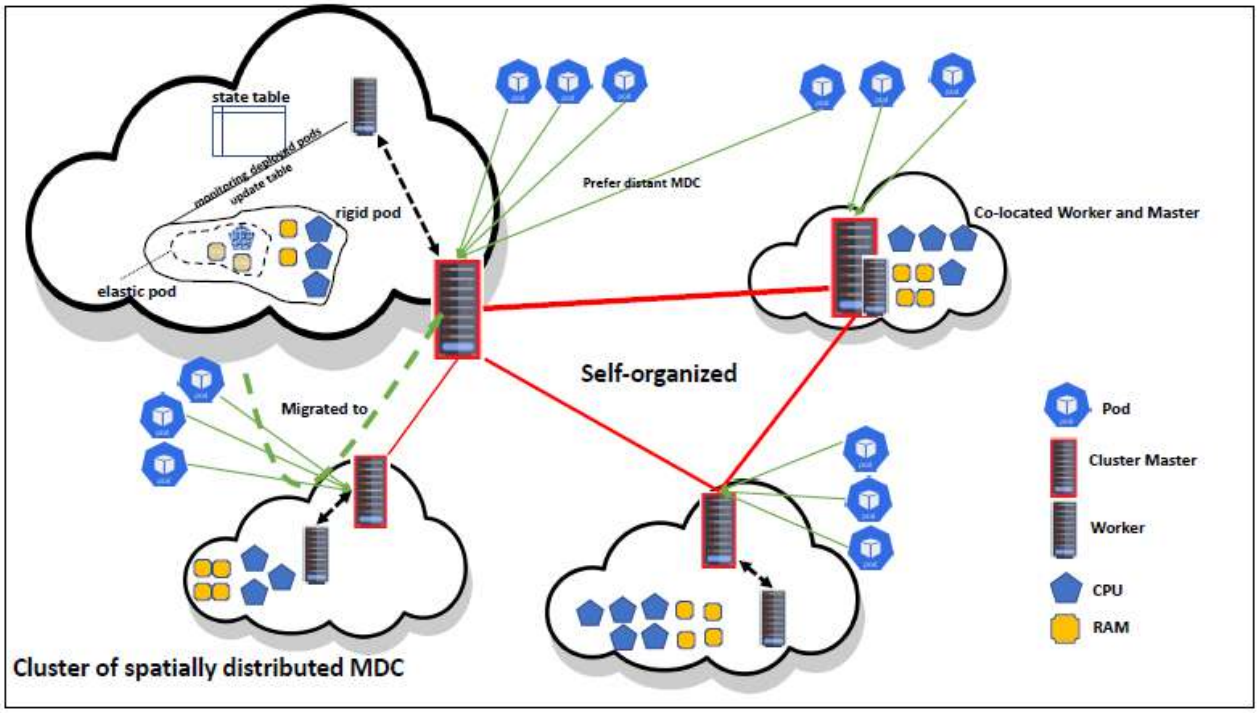

Figure 7: Schematic of a self-organized system of edge micro data center to orchestrate resource utilization: pods are submitted to each cluster's master, which may be forwarded to the same cluster worker agent or transferred to another cluster. The worker of each cluster manages the whole cluster resource pool and monitors

and updates the resource slack of current deployed rigid pods in the state table. Elastic pods may be assigned to a currently deployed rigid pod to utilize the slack resources.

Within each cluster, multiple worker agents are present, each equipped with a defined amount of CPU and RAM resources. While our model and solution primarily focus on these two resource dimensions, they can be extended to include additional resources such as storage and GPU units. A worker agent is responsible for accepting and executing pods assigned by the cluster master agent.

For simplicity, we assume that all available resources within a cluster are managed collectively by a virtual single worker, allowing us to focus on optimizing intra-cluster resource utilization. The worker continuously monitors the resource usage and slack of currently deployed pods, maintaining and updating a state table to reflect the current allocation.
Each worker agent has a specific CPU and RAM capacity, denoted as Wcap=(CPUcap,RAMcap). In our setup, we assume Wcap=(512,512) units.
We distinguish between two types of pods: rigid pods and elastic pods. Rigid pods have strict execution time requirements and are considered unsatisfied if their queuing time exceeds a predefined threshold.

Also, rigid pods may overestimate their resource demands, which incurs resource slack after deployment. Elastic pods, on the other hand, could tolerate a higher queuing delay before serving; the queuing delay tolerance is greater than rigid pods and does not overestimate their needs. The CPU and RAM demand and slack of each pod are denoted by a tuple D = (CPU demand,RAM demand) and S = (CPU slack,RAM slack) noting that the slack values for elastics pods are zeros. Each submitted pod's requested demand and possible slack follow a stochastic distribution with expectations E[D] and E[S].

An example with three pod profiles reflecting different types of applications can be seen in Figure 8.

Profile	**Demand** (CPU, RAM)	**Slack (if rigid)** (CPU, RAM)	**Demand steps** $\mathcal{U}(T_{min}, T_{max})$
Small	(1,1), (1,2), (2,1), (2,2)	(0, 0)	(60, 120)
Medium	(4, 4) (4,6) (6,4) (6,6)	(1,1), (1,2), (2,1) (1,2), (2,1), (2,2) (1,2), (2,1), (2,2) (1,2), (2,1), (2,2)	(100,200)
Large	(8, 8) (8,16) (16,8) (16,16)	(4,4) (4,4), (4,6) (4,4), (6,4) (4,6), (6,4), (6,6)	(150,300)

Figure 8: Pod profiles

4.3 Self-organized Solution Approach in the Emergent Scheduler

We model the system as a multi-agent environment comprising a master agent, a worker agent, and a sequence of arriving pod agents. Each agent possesses distinct attributes and operates according to its role at each step of the process. Initially, the system instantiates the master and worker agents. During each simulation step, the system scheduler invokes the step function for all agents and, based on a predefined schedule, may introduce a new pod agent into the system. The inter-arrival times for pods are generated according to an exponential distribution with parameter λ. The scheduler is responsible for tracking which pods are satisfied or unsatisfied, and it removes completed pods from the system as they finish execution.

4.3.1 Pod Agent Behaviour

Each pod agent is characterized by its type (either elastic or rigid), its resource demand, required execution steps, maximum acceptable queuing time, and any potential slack. Pods are placed in the master agent's queues-separated into rigid and elastic queues-and are then offered to the worker agent. When a pod is accepted by the worker, it becomes a deployed pod.
At each time step, a pod agent either waits in the appropriate master queue, decreases its remaining execution time if it is already deployed, or completes its execution. Upon completion, the system determines whether the pod's requirements have been satisfied. The system then releases the resources allocated to the pod in the worker agent and removes the pod from further scheduling.

4.3.2 Worker Agent Behaviour

The worker agent is responsible for monitoring CPU and RAM assignments and utilization at each time step. Its primary function is to accept or decline pods offered by the master agent, with its behaviour differing based on whether the pod is rigid or elastic.

Rigid Pods:
When offered a rigid pod, the worker simply checks if the pod's CPU and RAM demands can be met with the currently available resources. If sufficient resources are available, the worker accepts the pod, updates the resource assignment and utilization status, and proceeds with deployment. If not, the worker sends a decline message back to the master.

Elastic Pods:
For elastic pods, the worker selects a peer pod from the currently deployed pods using a specified selection algorithm. It then checks whether the chosen peer pod has enough slack resources (i.e., unused CPU/RAM) to accommodate the elastic pod's request. If the slack is sufficient, the worker assigns these resources to the elastic pod, thus improving overall resource utilization. If not, the worker attempts to accept the elastic pod as if it were a rigid pod. If this also fails, the worker declines the pod and notifies the master.
We assume that each peer pod can host only one elastic pod. Therefore, it is crucial to select the peer with slack resources that most closely match the elastic pod's CPU and RAM requirements. The main challenge lies in efficiently selecting the appropriate peer pod, especially given the scalability concerns and the dynamic nature of slack resources among deployed rigid pods. Exhaustively searching for the best match among all deployed pods is impractical. To address this, we implement three different algorithms for peer selection.

4.3.2.1 Random Selection

In this approach, the worker agent randomly and uniformly selects one of the currently deployed pods and checks if it can accommodate the elastic pod. While this algorithm is highly scalable due to its simplicity, it often fails to find an optimal match for the elastic pod's resource requirements.

4.3.2.2 Best Match

Serving as a benchmark, the best match algorithm exhaustively evaluates all currently deployed pods, calculating a matching score for each candidate that has sufficient resources. The dis-matching score is defined as the sum of the absolute differences in CPU and RAM between the elastic pod and the candidate pod. In the event of a tie, the candidate whose remaining execution steps most closely match the elastic pod's required execution steps is chosen. This dual-criteria approach aims to preserve pods with higher slack for future requests; while also prioritizing candidates whose slack most closely fits the elastic pod's needs. Ideally, the best match is a rigid pod with slack resources exactly matching the elastic pod's demand and whose remaining execution time aligns with the elastic pod's required duration. Importantly, if the selected host pod remains in the system after the elastic pod completes, it cannot share its slack resources with other pods.

4.3.2.3 ABC-based Bottom-up Resource Orchestration

Our proposed algorithm is inspired by the artificial bee colony algorithm, a swarm intelligence technique originally introduced by Karaboga . In this agent-based approach, pods are analogous to bees searching for food sources-that is, available resources. Each rigid pod (bee) monitors its available slack and updates its "food quality" for incoming elastic pods. The worker agent maintains a compact lookup table that groups currently deployed pods into buckets based on their slack resources. We define two levels for both CPU and RAM slack (small and large), resulting in four possible buckets: LL, LH, HL, and HH (e.g., LL represents pods with low CPU and low RAM slack). Pods are assigned to these buckets according to their observed slack, and may be reassigned as their slack changes, although this dynamic reassignment is not considered in this version.

When an elastic pod arrives, the worker selects the appropriate bucket based on the pod's resource demand and then randomly chooses a peer pod from that bucket. This method combines the scalability of random selection with the advantage of utilizing prior knowledge about the slack resources of rigid pods, thereby improving the likelihood of a suitable match while maintaining efficiency.

4.3.3 Master Agent Behaviour

The master agent is responsible for maintaining and updating both the rigid and elastic pod queues. At each time step, the master prioritizes serving rigid pods, processing as many as possible. It does this by repeatedly fetching rigid pods from the rigid queue and offering them to the worker agent until a decline message is received. Once no more rigid pods can be served, the master turns its attention to the elastic queue.
For elastic pods, the master fetches the pod at the front of the queue and offers it to the worker as an elastic pod. If the worker declines the elastic pod, the master may attempt to serve it as a rigid pod with probability
Γ. . This means that, on average, every $1/\Gamma$ rounds, the master gives elastic pods the opportunity to use raw resources as if they were rigid pods. The parameter Γ serves as a control to balance the satisfaction rates between rigid and elastic pods. Our results show that with appropriate tuning of Γ , the satisfaction rate for elastic pods can be improved without negatively impacting the satisfaction rate of rigid pods.

Algorithm 1: Master Agent Behavior

```
Main step()
    while len(rigid_queue) > 0 and next_rigid_pod() do
        Continue;
    while len(elastic_queue) > 0 and next_elastic_pod() do
        Continue;
Function next_rigid_pod()
    Fetch the next pod from the rigid_queue;
    if worker accepts as rigid then
        Remove pod from queue;
        Update state table of deployed pods;
        return True;
    else
        return False;
Function next_elastic_pod()
    Fetch the next pod from the elastic_queue;
    if worker accepts as elastic then
        Remove pod from elastic queue;
        Update state table of deployed pods;
        return True;
    rnd = random number in (0,1);
    if rnd < Γ and worker accepts as rigid then
        Remove pod from queue;
        Update state table of deployed pods;
        return True;
    else
        return False;
```

Figure 9: Pseudocode of the master agent behaviour

4.4 NN-based Peer Selection

Recent advances in machine learning have significantly improved resource allocation in edge computing, challenging the effectiveness of traditional centralized strategies. Building on this progress, our work enhances the bottom-up framework by introducing a neural network-based peer selection mechanism. Specifically, we employ a two-hidden-layer MLP trained on data generated by the best-performing algorithm from Section 4.3. This intelligent peer selection replaces the previous random approach, enabling more effective resource allocation while preserving the decentralized nature of the system.

Our research demonstrates that this integration not only boosts scheduling efficiency but also maintains the agility and scalability essential for edge environments. By combining ABM with machine learning, we create a more adaptive edge orchestration system. This fusion of learning-based techniques with ABM bridges the gap between theoretical scheduling strategies and practical deployment, marking a significant step forward in adaptive and efficient edge resource management.

4.4.1 Methodology

The original bottom-up resource orchestration framework from Section 4.3 offers a computationally efficient solution for resource allocation in edge environments. In this framework, the edge node categorizes deployed rigid pods into buckets based on their available slack resources. Specifically, when considering CPU and memory slack, four buckets are defined: (LL), (LH), (HL), and (HH), where

L and H denote low and high slack, respectively. The node maintains a lookup table for these buckets, each containing the corresponding deployed pods.

When an elastic pod arrives, the system selects the appropriate bucket based on the pod's resource demand and then chooses a peer rigid pod uniformly at random from that bucket. This method is highly scalable and leverages initial slack estimates for more informed assignments. However, while computationally lightweight, the random selection process does not fully exploit the potential for near-optimal resource allocation decisions.
To address this limitation, our work enhances the resource allocation strategy by integrating a NN-based peer selection mechanism. The NN model is trained on a dataset generated by the Best algorithm, which computes the optimal mapping of elastic pods to rigid peers using complete system information.

The Best algorithm creates training data by evaluating the fitness of all potential peer pods for each incoming elastic pod. This exhaustive approach is computationally intensive, as it scans the entire list of deployed rigid pods to identify the optimal match. The fitness evaluation considers factors such as resource compatibility and execution time alignment, ensuring that the training data reflects the most effective allocation decisions.

Resource Match (f1): This metric quantifies the difference between the slack resources available in a deployed rigid pod and the resource demand vector of the incoming elastic pod.

Temporal Match (f2): This metric measures the difference between the remaining execution steps of the deployed rigid pod t_exer and the required execution steps of the elastic pod t_exee.
Together, these metrics provide a comprehensive assessment of the suitability of each peer rigid pod for hosting an elastic pod. The training dataset incorporates features such as the demand vector of the incoming elastic pod, the slack resources of each deployed rigid pod, and the computed fitness scores (f1, f2) for every rigid-elastic pod pair. The pod arrival rate, denoted by λ, is varied to evaluate system performance under both light and heavy load conditions.

4.4.1.1 Neural Network Training

For peer selection, we employ a MLP model with two hidden layers containing 64 and 32 neurons, respectively. The model utilizes the ReLU activation function and is trained using MSE as the loss function. Training is performed on a normalized dataset with the following features:

- CPU and memory demand of the incoming elastic pod
- Slack resources of deployed rigid pods
- Execution steps required by the elastic pod and remaining steps for each deployed rigid pod

The MLP's output predicts the fitness scores (f1, f2) for each elastic–rigid pod pair, enabling more informed and effective resource allocation decisions.

4.4.1.2 Peer Selection Process

The peer selection procedure is outlined in Figure 10. During runtime, the trained MLP model (referred to as nn_model) predicts the fitness scores of all potential peer pods (candidate peers) for each incoming elastic pod (new_pod). The rigid peer pod with the lowest predicted f1 score is chosen as the best match. If multiple candidates have similar f1 scores, both f1 and f2 are considered to make the final selection, ensuring an optimal pairing.

Input: List of deployed rigid pods $candidate\_peers$, incoming pod new_pod, trained neural network nn_model
Output: Best peer pod $best\_peer$
$best\_peer \leftarrow$ None
$current\_pods \leftarrow$ Deployed pods in $candidate\_peers$
if $current\_pods = \emptyset$ **then**
  return None
end
for *each pod in* $current\_pods$ **do**
  Prepare input vector $X \leftarrow$ [new pod demand, peer pod slack, demand steps]
  Normalize X
  Predict fitness $[f1, f2] \leftarrow$ nn_model.predict(X)
  Store $peer\_fitness[pod] \leftarrow [f1, f2]$
end
$best\_peer \leftarrow$ Pod with min $f1$ (and $f2$, if required)
return $best\_peer$

Algorithm 1: NN-Based Peer Pod Selection Algorithm

Figure 10: Pseudocode for the peer selection procedure

4.5 Results

We implement the system simulation as a multi-agent environment using the MESA library in Python. Over the course of 12000 time steps, we monitor key metrics such as worker CPU/RAM utilization and the lengths of the master's queues. Our initial analysis focuses on the dynamics of the master's queue lengths under varying traffic intensities. To classify resource levels, we use: i) a threshold of 5 units for both CPU and RAM to distinguish between low and high resource clusters, and ii) the proposed NN-based peer selection mechanism.

Figure 11-(a) illustrates that under light load conditions ($\lambda \approx 0.5$), queue lengths fluctuate but do not accumulate over time, and worker utilization remains similar across all algorithms since available resources exceed demand. As λ increases and demand surpasses available resources, the system enters a heavily loaded regime. In Figure 11-(b) ($\lambda \approx 0.8$), queue lengths consistently build up, and resource utilization becomes dependent on each algorithm's ability to efficiently exploit slack resources.
Figure 12 presents the CPU/RAM utilization and the satisfaction rates for both rigid and elastic pods, which serve as key performance indicators. When λ is below a critical threshold (λ_0), the satisfaction rate approaches one. Beyond this threshold, satisfaction rates are determined by how effectively each algorithm leverages slack resources. The proposed bottom-up algorithm improves overall utilization and enhances pod satisfaction rates, performing between the random and best-match strategies as anticipated.
It is important to highlight that the inherent randomness of the bottom-up approach makes it robust against inaccuracies in slack resource estimation or threshold settings.

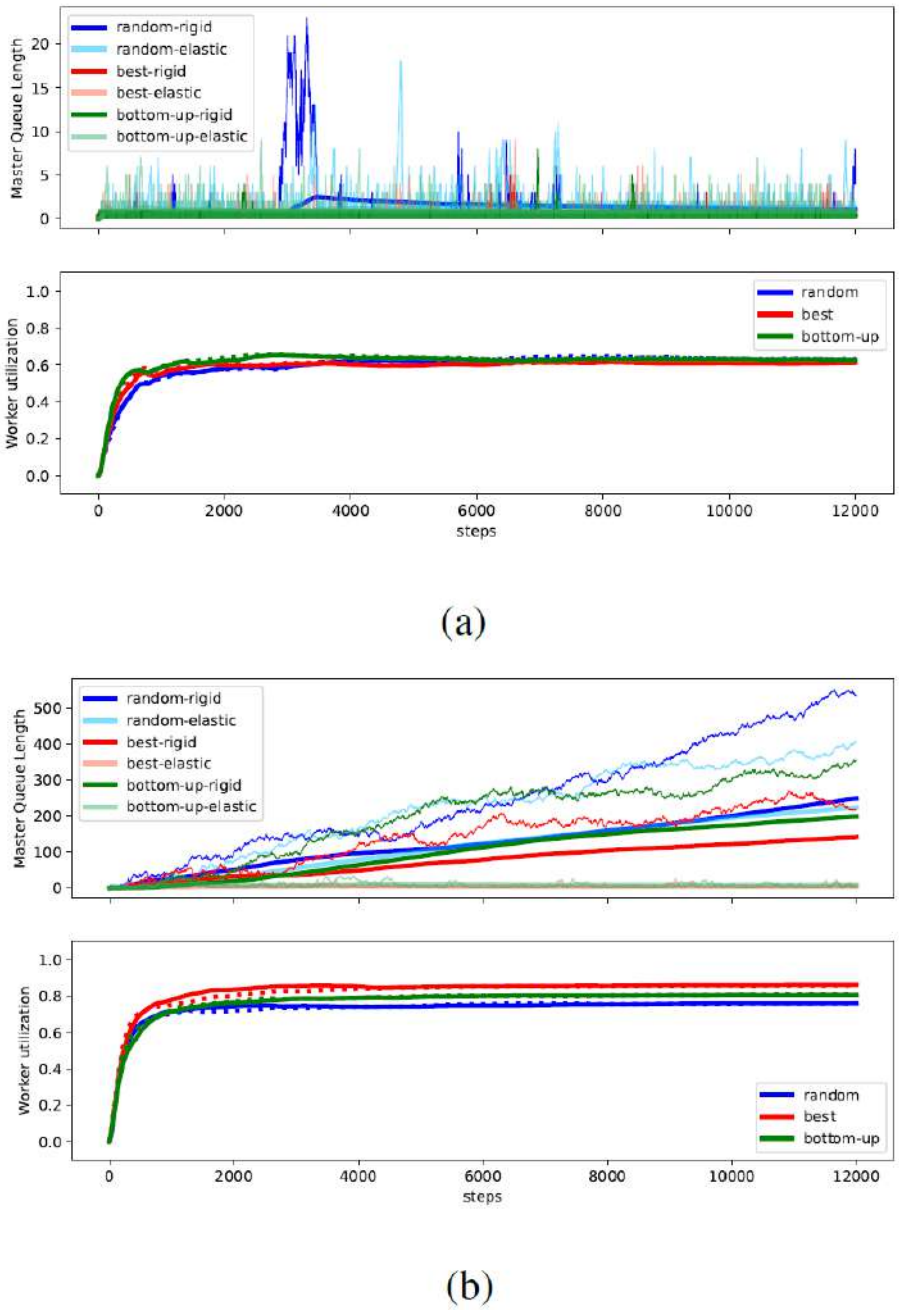

Figure 11: Queue length dynamics for (a) λ = 0.5, lightly loaded regime, (b) λ = 0.7, heavy loaded regime. The thin curves show the instantaneous queue length, and the solid curves show the running average

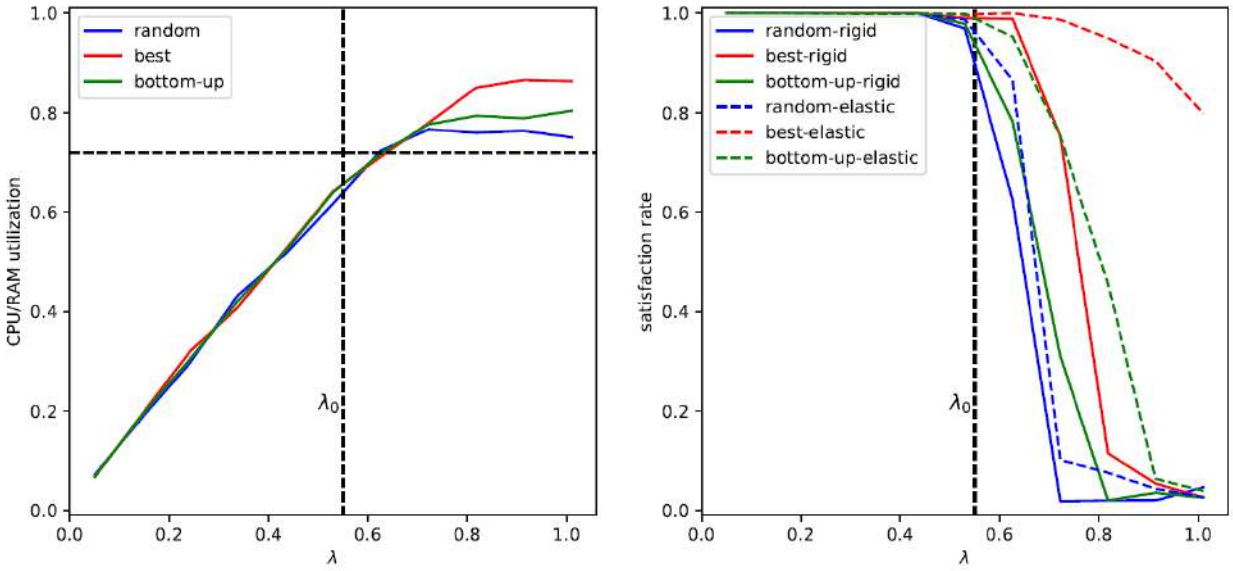

Figure 12: The utilization and satisfaction ratio against different values of λ where the cluster label of 20% of deployed pods is selected randomly

The performance of the Bottom-up with NN algorithm is compared against the three baseline algorithms described before. As illustrated in Figure 13, the Best algorithm consistently outperforms the others in both resource utilization and satisfaction rate.
Simulation results indicate that the NN-basic algorithm, when trained on a limited dataset of 2000 samples, performs worse than random peer selection. However, increasing the training set to 12000 data points enables the NN-basic algorithm to achieve results comparable to the original Bottom-up approach.

The original Bottom-up and NN-basic algorithms show similar performance across the evaluated metrics. In contrast, the Bottom-up with NN algorithm delivers significant improvements: by leveraging the NN for peer pod selection within resource buckets, rather than relying on random selection, it achieves resource utilization much closer to that of the Best algorithm, as seen in Figure 13.

Notably, the Bottom-up with NN approach also yields a marked increase in the satisfaction rate, particularly for elastic pods, without compromising the satisfaction of rigid pods. This demonstrates the effectiveness of integrating neural network-based decision-making into the Bottom-up framework for adaptive and efficient resource allocation.

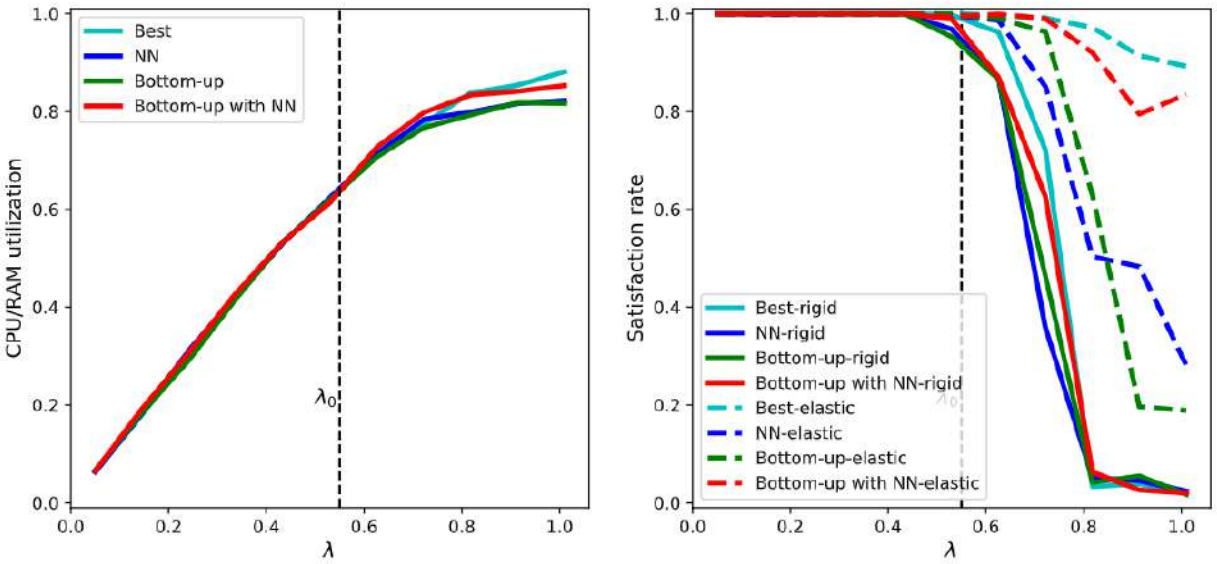

Figure 13: Performance comparison for different algorithms in terms of resource utilization versus arrival rate (left), and satisfaction rate versus arrival rate (right)

5 Multi-Agent AI Algorithm for Decentralized Resource Allocation

5.1 Introduction

In the ACES distributed cloud systems, the efficient allocation of workloads across computing nodes is a fundamental requirement to ensure responsiveness, resource utilization, and service quality. To address this need, the swarm-based algorithm (see Section 4.3) has been adopted to perform decentralized workload allocation decisions, leveraging local information and interactions among agents to achieve global coordination. However, the behaviour and effectiveness of the swarm algorithm are governed by a set of hyperparameters that must be carefully calibrated to balance exploration, stability, and convergence speed. Manual tuning or heuristic approaches are commonly used in practice but are often inadequate in dynamic environments where system conditions and user demands vary over time. To overcome these limitations, a machine learning-based mechanism has been implemented to enable the automatic calibration of the swarm algorithm's hyperparameters, allowing the system to adapt its behaviour in a data-driven manner during execution.

Given the complexity of the swarm-based algorithm used for workload allocation, its performance critically depends on a set of hyperparameters whose optimal values are not analytically known. Thus, optimizing the swarm algorithm can be viewed as a **black-box global optimization problem**, in which evaluating the objective function—i.e., assessing the swarm's performance under specific hyperparameter configurations—is computationally expensive and must be performed via simulations or experimental testing. Active learning algorithms are particularly suitable for addressing this class of problems, as they aim to efficiently find optimal solutions with minimal function evaluations by iteratively building surrogate models and intelligently selecting new query points.

5.2 State-of-the-art in black-box optimization

Among existing approaches, BO [14] represents the state-of-the-art active learning method, leveraging probabilistic surrogate models (typically Gaussian Processes) to approximate the unknown objective and using acquisition functions that balance exploration and exploitation. Other related active learning schemes with similar principles have also been proposed, including methods based on radial basis functions or ensemble models [15-17]. However, these methods typically rely solely on probabilistic arguments to handle exploration. To enhance this exploration capability deterministically, the approach considered here integrates an IDW strategy into the active learning scheme.
Specifically, the proposed algorithm, named D-GLIS (Distributed Global optimization via Local surrogate and Inverse distance weighting Sampling) [16], extends the recently introduced GLIS approach by combining radial basis function surrogate models with a deterministic IDW function. The IDW term explicitly encourages the exploration of unexplored regions of the feasible space, complementing the probabilistic approach typical of BO.
Moreover, differently from centralized methods such as BO, D-GLIS is inherently decentralized, specifically designed for optimization scenarios in distributed multi-agent networks. In D-GLIS, agents independently construct and maintain local surrogate models and optimize local acquisition functions without sharing their local objectives or surrogates. Cooperation among agents is realized through decentralized consensus mechanisms relying only on limited information exchange. This decentralized and deterministic exploration structure makes D-GLIS particularly advantageous in scenarios characterized by dynamic topologies, limited communication bandwidth, privacy constraints, or robustness requirements, common in distributed cloud systems.

5.3 Methodology

5.3.1 Problem Formulation

Consider a network consisting of NN computational units or agents. The goal is to solve the optimization problem:

$x^* = \arg\min_{x \in X} \quad f(x)$,

where $x \in \{R\}^n$ n and $X \subseteq \{R\}^n$ is a known feasible set. The global objective $f(x)$ is assumed to be separable:

$f(x) = \Sigma_{i=1}^{N} f_i(x)$, ,

where each f_i: $\{R\}^n \to \{R\}$represents the local cost function evaluated solely by agent i. Specifically, it is assumed that:

- Each local cost function $f_{i(x)}$ lacks an explicit analytical expression and can only be assessed through direct evaluation at points $x \in X$, possibly affected by noise.
- Agents independently evaluate their respective functions $f_{i(x)}$ without sharing raw evaluation results yet collaborate toward the global objective.
- Evaluations of $f_{i(x)}$ are computationally costly, necessitating optimization methods that minimize the number of evaluations.
- Agents communicate over a fixed, strongly connected, and doubly stochastic weighted graph $\{G\}$, where nodes represent agents and edges define possible communication links.

5.3.2 Local Surrogate Functions

Each agent i maintains a local dataset $D_i = (x_1, y_1), (x_2, y_2), \ldots, (x_{M_i}, y_{M_i})$, where each observation $y_j$approximates the local objective$f_i(x_j)$. This dataset remains private to the agent. Each agent i constructs a local surrogate function $\hat{f}$using an RBF approach:

$\widehat{f_i}(x) = \Sigma_{k=1}^{M_i} \beta_k^{(i)} \phi(\varepsilon d(x, x_k))$,

where $\phi(\cdot)$ denotes an RBF,$d(\cdot,\cdot)$ is a chosen distance metric, and ϵ a hyperparameter controlling the shape of the RBF. The coefficients$\beta_k^{\{(i)\}}$ are identified by minimizing a regularized least-squares cost:

$\Sigma_{k=1}^{M_i} \left[y_k - \Sigma_{j=1}^{M_i} \beta_j^{(i)} \phi\left(\varepsilon d(x_k, x_j)\right) \right]^2 + \gamma \left\| \beta^{(i)} \right\|^2$,

ensuring numerical stability and convexity. Common RBF choices include inverse quadratic and squared exponential kernels.

Each agent independently constructs a local surrogate $\widehat{f_{i(x)}}$, and these local surrogates combine to form the global surrogate:

$\hat{f}(x) = \Sigma_{\{i=1\}}^{N} \widehat{f(x)}$.

However, as agents do not share surrogates, distributed optimization techniques are required to minimize $\hat{f}(x)$.

5.3.3 Local Inverse Distance Weighting Functions

To promote exploration, each agent employs an IDW function, defined as in Figure 14 below:

$$z_i(x) = \begin{cases} 0 & x \in \{x_1, \ldots, x_{M_t}\} \\ \tan^{-1}\left(\frac{1}{\sum_{j=1}^{M_t} w_j(x)}\right) & \text{otherwise} \end{cases}$$

Figure 14: IDW expression for agent i

This function ensures higher values for regions less explored by agent i.

5.3.4 Local Acquisition Functions

Each agent constructs a local acquisition function $a_i(x)$ balancing exploration and exploitation:

$a_i(x) = \frac{\hat{f}(x)}{\Delta\hat{f}} - \delta z_i(x)$,

where $\Delta\hat{f}$ is the range of the surrogate $\hat{f}$and normalizes it, and $\delta \in (0,1]$ controls exploration intensity. At each iteration, agent ii, selected in a round-robin fashion, solves:

$x^{i,*} = \arg\min_{x \in X} \quad a_i(x)$,

using distributed optimization.

5.4 Iterative Optimization Process

Upon selecting $x^{i,*}$, agent i evaluates $y_i = f_i(x^{i,*}) + \epsilon$, updates its dataset and surrogate, and the process repeats with the next agent until reaching a predefined iteration limit. The final global solution is obtained through distributed minimization of the global surrogate without the exploration term. The overall procedure si summarized in Figure 15 below.

Algorithm 1 D-GLIS

Inputs: maximum number of function evaluations per agent $N_{\max}$; exploration parameter δ; constraint set $\mathcal{X}$; initial datasets $\mathcal{D}_i = \{x_j, y_i\}_{j=1}^{M_i}$, and initial surrogate functions $\hat{f}_i$ -constructed from $\mathcal{D}_i$- for all agents $i = 1, \ldots, N$.

1: **repeat**
2: select agent i according to a cyclic rule
3: build the IDW function z_i in Section (III-B) from $\{x_j\}_{j=1}^{M_i}$
4: define the local acquisition function a_i in (6)
5: compute (7) via distributed optimization (see Section IV)
6: evaluate $y^{i,*} = f(x^{i,*}) + \epsilon$
7: update the local dataset $\mathcal{D}_i \leftarrow \mathcal{D}_i \cup \{x^{i,*}, y^{i,*}\}$
8: $M_i \leftarrow M_i + 1$
9: fit a new local surrogate $\hat{f}_i(x)$ based on $\mathcal{D}_i$
10: construct the global surrogate $\hat{f}(x)$ as in (5)
11: **until** maximum numbers of iterations $T_{\max} = NN_{\max}$ is reached
12: compute consensus $x^* = \arg\min_x \sum_{i=1}^{N} \hat{f}_i(x)$ via distributed optimization (see Section IV)

Output: consensus x^*

Figure 15: Psuedo-code for the D-GLIS algorithm

5.4.1 Distributed Optimization Technique

Optimization of the local acquisition functions is performed through distributed algorithms since each agent lacks direct knowledge of other agents' surrogates. Specifically, the GTAdam algorithm [18] is utilized—a distributed variant of the Adam optimization method [19]—integrating gradient tracking for consensus-driven convergence. GTAdam relies solely on local computations and neighbor-to-neighbor communication, ensuring suitability for decentralized environments. Although convergence guarantees are limited by the non-convex nature of the optimization, GTAdam effectively supports cooperative minimization tasks within D-GLIS.

The GTAdam approach is employed in two key stages of the D-GLIS algorithm: the iterative minimization of local acquisition functions (inner loop), and the final global surrogate minimization (outer loop).

5.5 Results

The algorithm has been implemented in Python, and all the codes are available in the ACES GitHub repository [20]. The disropt library [21] has been used for the distributed operations.
The DGLIS algorithm outlined in Section 4 was employed to calibrate the hyperparameters α and β of the swarm-based optimization framework described in Section 3. Both parameters were subject to box constraints within the interval [0,5]. A network consisting of 10 agents was simulated, and each experiment was replicated over 20 independent MonteCarlo runs. In all the experiments the agents communicate over a fixed undirected graph G, generated using an Erdos-Renyi random model (2, p) with p = 0.3. The adjaceny matrix of the graph is obtained through a Metropolis-Hastings weight model [22]. In every repetition, each agent was initialized with 4 points sampled uniformly at random from the feasible domain. The inner solver, GTAdam, was executed for 1000 iterations, and the parameter δ was configured based on a heuristic rule: $\delta_i = 10\left(\max_{y_j \in D_i} \quad y_j - \min_{y_j \in D_i} \quad y_j\right)$. The Figure 16 reports, at each iteration of the DLGIS procedure, the corresponding optimal point x^* that would be identified by the algorithm in the absence of the IDW term, if the execution were stopped at that specific iteration. In all simulations, the regularization parameter λ was set equal to 0.5. As illustrated by the figure, satisfactory configurations of α and β are consistently identified after approximately 30 iterations.

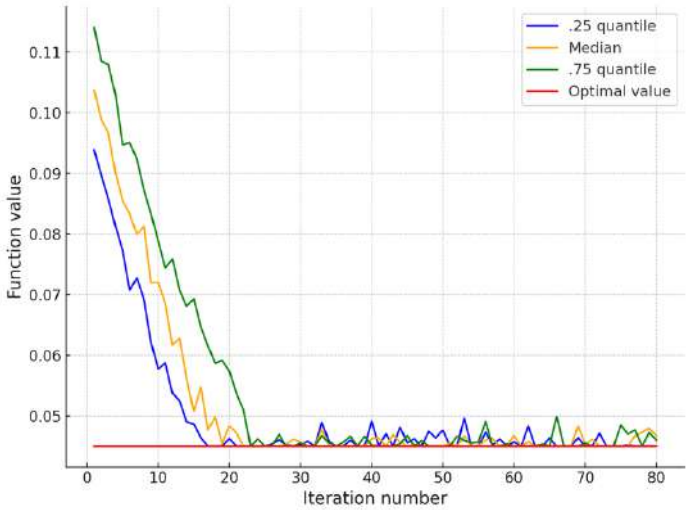

Figure 16: Performances of D-GLIS on self-hyperparameter calibration: function value vs. number of iterations

6 Distributed Data Management

6.1 Introduction

In ACES, distributed data management play an important role in supporting autonomy, system resilience, and the ability to adapt to changing conditions. Since edge-cloud infrastructures involve many locations where data is produced and consumed, it's essential to manage this data in a way that supports low-latency operations, high availability, and reliable coordination. Unlike traditional cloud setups that often relies on centralised data processing, edge environments are more fragmented and usually has limited resources. Data should ideally be processed close to where it originates but still be accessible across the system when needed. This raises some challenges around data placement, synchronisation, and consistency.

To address these issues, ACES uses a distributed and policy-driven layer for managing data. This layer works together with cognitive agents that are capable of reasoning about the system state and taking decisions in real time — such as where data should be stored, how it should be accessed, or when it needs to move. This chapter gives a conceptual overview of how distributed data management is approached in ACES. It outlines general ideas, technologies, and practices, and explains how these are used to support intelligent and self-managing behaviour in the system.

6.2 State-of-the-art in Distributed Data Management

Distributed data management is about how data is stored, moved, and accessed across different systems. In typical cloud setups, this is usually handled with centralized storage and strong consistency. But in edge-cloud environments, conditions are different. Data is often created and used locally, where resources are limited and connections may be unstable. The goal is to keep data close to where it's needed while still allowing coordination between parts of the system. Some key principles guide this. Data locality reduces delays and saves bandwidth. Replication helps ensure availability if a node fails. Full consistency is often too costly at the edge, so systems use eventual consistency instead. Partitioning helps to spread data and scale out. Technologies like HDFS [23] and Apache Ozone [24] offer scalable storage. Object stores like MinIO [25] are common for flexible data access. IPFS [26] and CRDTs support more advanced models like content-addressing or coordination-free replication.

Edge systems face unique difficulties. Devices may have very little storage or compute power. Networks can be unreliable. Data privacy rules may prevent data from leaving the device. The environment is also highly diverse in terms of hardware and software. A common response is to use hybrid approaches. These combine local storage with metadata that guides when and how to sync data. Metadata catalogs track locations, usage, and policy rules. This allows systems to make decisions about moving or deleting data without central coordination.

ACES builds on these ideas. It adds swarm-like cognitive agents that use telemetry and metadata to decide where to place workloads and how to manage data. These agents are adaptive and respond to system state in real time, which helps move beyond fixed, rule-based models and makes the platform more flexible and responsive.

6.3 ACES Data Management Principles

ACES follows a data-centric approach to support cognitive and autonomous behaviour at the edge. The system is designed to make data available for agents where and when it is needed, without relying on centralized storage and control. This is achieved through a combination of decentralized storage and intelligent data routing.

A key feature of ACES is the use of OpenTelemetry [27] to collect telemetry data across the edge-cloud infrastructure. OpenTelemetry gathers both infrastructure-level metrics – such as CPU, memory, disk I/O, and network usage – from Kubernetes nodes, as well as detailed telemetry from user workloads. This data is important for understanding system state, guiding workload placement, and enabling self-optimizing behaviour.

Core data management principles in ACES:

- **Local-first storage**: Data is stored and processed as close as possible to where it is generated. This reduces latency and limits unnecessary data transfers.
- **Metadata-driven decisions**: Every data object is associated with rich metadata, including its origin, type, sensitivity, and usage history. This metadata is used by cognitive agents to make informed decisions about data placement and retention.
- **Policy-based control**: Data movement and replication follow predefined policies. These policies can be based on SLA constraints or user preferences, defined by performance targets such as latency, throughput, data retention duration, or energy consumption thresholds, as well as regulatory or business-specific compliance requirements.
- **Event-driven processing**: ACES supports reactive data flows. For example, a sudden spike in CPU usage observed via OpenTelemetry can trigger redistribution of workloads or replication of data to prevent overload.
- **Consistency vs. availability trade-off**: In many cases, ACES opts for eventual consistency to ensure high availability and responsiveness. Synchronization with peer nodes happens asynchronously, depending on network conditions and operational priorities.

The collected telemetry data plays a foundational role. It is continuously processed by local agents and aggregated into the DKB, which serves as the shared memory of the system. This data helps agents evaluate the state of the platform and take autonomous actions, such as offloading data to another site or adapting storage configurations. In ACES, data is not passive. It actively shapes the behaviour of services. The combination of OpenTelemetry, decentralized data control, and agent-based reasoning enables ACES to meet its goals of autonomy, resilience, and performance in edge-cloud environment.

6.4 Standards

Telemetry collection in ACES uses the OpenTelemetry, ensuring compatibility with industry tools and enabling consistent metrics across heterogeneous infrastructure. For data storage and movement, ACES explores integration patterns inspired by widely used technologies such as POSIX-compliant file systems, object storage APIs (via MinIO), and Kubernetes-native interfaces. Standardization also applies to metadata management. The potential use of a global metadata catalogue via Nuvla ensures discoverability and traceability while respecting data sovereignty and edge autonomy.

These choices help ACES to remain open, interoperable, and adaptable to various deployment scenarios without locking into proprietary protocols and tools.

6.5 Operational Insights into Edge-to-Edge Data Management

The ACES project builds on a growing body of experience from previous edge-cloud initiatives in which project partners participated previously, which provide valuable lessons on how to design, deploy, and operate distributed data systems across heterogeneous and dynamic edge infrastructures.

In some of the EU and ESA projects, Nuvla/NuvlaEdge platform was used to orchestrate workloads and manage data flows across edge sites (for more details see series of technical blog posts in [30]). The platform supported real-time analytics and AI workloads on mobile and fixed assets – such as trams, warehouses, and roadside units – without relying on centralized cloud storage. Instead, the architecture used edge-local processing and peer-to-peer data exchange between sites.

One key insight is that maintaining data locality while ensuring global coordination is best achieved by separating data payloads from metadata. For example, Nuvla-based deployments used local storage for time-sensitive data and a global metadata registry for coordination. This approach allowed edge services to operate autonomously, while remaining discoverable and controllable through the central management plane (see Figure 17).

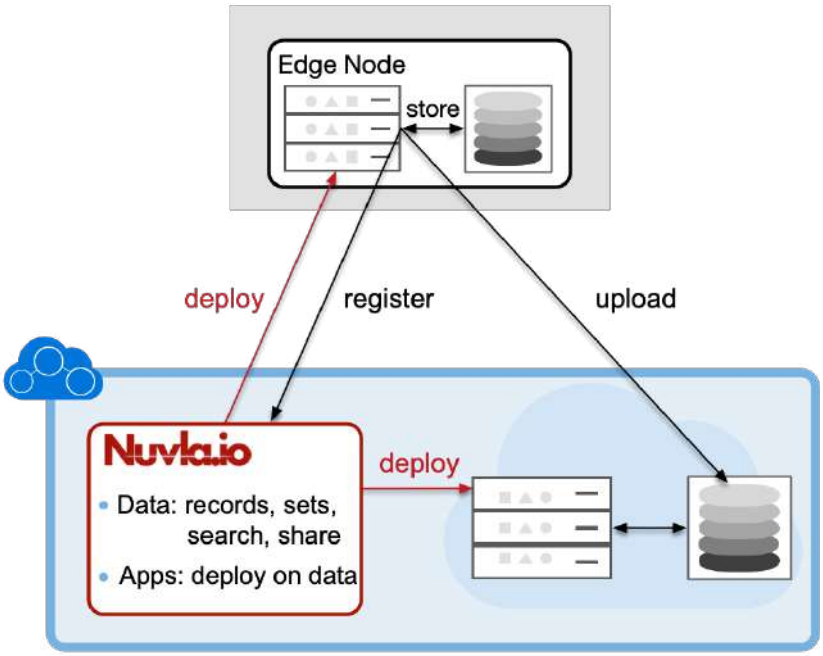

Figure 17: Data records, objects, and sets represent the metadata catalogue on Nuvla.io. Augmented with powerful search, as well as objects and metadata sharing capabilities

In these deployments, data flow patterns were driven by workload requirements, network conditions, and device capabilities. For instance, sensor data collected at the edge was pre-processed locally and shared laterally with neighbouring nodes. Only essential summaries or decisions were forwarded upstream. This model aligns well with ACES goals of autonomy, low latency, and resilience. Typical topologies involved multiple edge nodes forming a mesh. Each node could operate independently or collaborate with peers, depending on service needs and resource availability. These edge-to-edge configurations supported flexible workload placement and adaptive data routing without compromising on performance or privacy.

These operational insights demonstrate the feasibility and advantages of decentralized data handling in real-world environments. They also show how agent-based orchestration, edge-local processing, and metadata-driven coordination forms a practical foundation for cognitive, autopoietic systems like ACES.

6.6 Data Management and Storing Implementation

The ACES project addresses distributed data management as a core enabler of its cognitive edge-cloud architecture, responding to the increasing complexity and volume of data across highly heterogeneous and geographically dispersed environments. Distributed data in ACES is collected from a variety of sources, including IoT devices, edge clusters, microservices, and centralized cloud environments, and managed through a layered, resilient architecture designed to ensure data accessibility, consistency, and semantic integrity.

Data in this context is categorized into metrics (e.g., CPU load, memory usage), real-time streams (e.g., alerts), graph-based relationships, and general object storage. Each data type is handled by distinct storage solutions: TimescaleDB [28] for time-series data, Neo4j [29] for graph structures, and MinIO

for scalable object storage. The system employs a push-pull telemetry model, combining real-time data pushing (for ephemeral or dynamic sources) with polling mechanisms for more persistent services, ensuring that no critical insight is lost.

To support interoperability and data fusion, the ACES Data Model adopts a graph-based representation aligned with NGSI-LD, enabling the transformation of raw telemetry into semantically enriched knowledge. The system utilizes NATS-based pipelines for decoupling data ingestion and consumption, allowing for scalable processing and aggregation. This facilitates seamless integration of telemetry streams from distributed sources and allows for flexible aggregation and real-time processing workflows. Once processed, data feeds into the CF, where machine learning and swarm intelligence models enable proactive orchestration and scheduling decisions, especially in multi-cluster settings.

One of the distinguishing features of ACES' approach is the DKBRS, which supports consistency and coordination across clusters. This component facilitates synchronized state awareness, enabling informed cross-cluster decisions by sharing knowledge artifacts like metrics, alerts, and predictions.

ACES also tackles data retention and lifecycle management through the use of ETL pipelines orchestrated with Prefect. These pipelines offload historical data to long-term storage, maintaining system performance while preserving analytical value. Altogether, this distributed data management strategy ensures the system's resilience, agility, and intelligence core to the autopoietic vision of adaptive edge-cloud services.

Technology	Primary Role	Application in ACES	Key Strengths
TimescaleDB	Time-series data storage	Stores fine-grained telemetry metrics (e.g., CPU, memory usage)	SQL compatibility, optimized for time-series data
Neo4j	Graph-based data storage	Stores relationships among pods, nodes, metrics	Graph modelling, semantic querying, relationship-aware
MinIO	Object storage (S3-compatible)	Long-term archival of old metrics via retention pipelines	Scalable, lightweight, S3-compatible
NATS	Real-time messaging and stream pipeline	Decouples telemetry producers/consumers, real-time alerts	High-throughput, fault-tolerant, scalable
Kubernetes	Container orchestration	Deploys, scales, and manages microservices across clusters	Automated scaling, resilience, CI/CD integration

Prefect	Workflow orchestration (ETL pipelines)	Coordinates retention flows to MinIO, manages metrics offloading	Python-native, robust task scheduling
CF	AI/ML-based analytics and orchestration	Consumes data for intelligent scheduling via swarm intelligence	Model serving, real-time decision making
DKBRS	Cross-cluster knowledge synchronization	Ensures state consistency between distributed clusters	Federated data sharing, consistency management
Prometheus	Monitoring and metric collection	Gathers system-level metrics (used in pull-push model)	Widely adopted, integrates with Kubernetes

Table 7: Comparative Table of technologies in ACES distributed data management

6.7 Cognitive framework

With data distributed across the ACES Platform and stored in a variety of technologies-from TimescaleDB for time-series data, to Neo4j for graph data, and NATS for real-time messaging, it becomes increasingly important to establish a unified entry point for data access. This central gateway is essential because most components within ACES rely on timely and consistent data to function effectively, whether they are training machine learning models, generating real-time alerts to predict critical system behaviour, or executing orchestration tasks. Without a universal access layer, integrating and managing these diverse data sources would be complex and inefficient, potentially hindering the platform's ability to deliver reliable and intelligent services.

Developed by HIRO, the CF serves as the central data orchestration layer for the ACES Platform, enabling unified access to distributed resources. As illustrated in Figure 18, the CF acts as the primary gateway for all data operations, abstracting the complexity of interacting with heterogeneous storage systems (TimescaleDB, Neo4j, NATS, etc.).

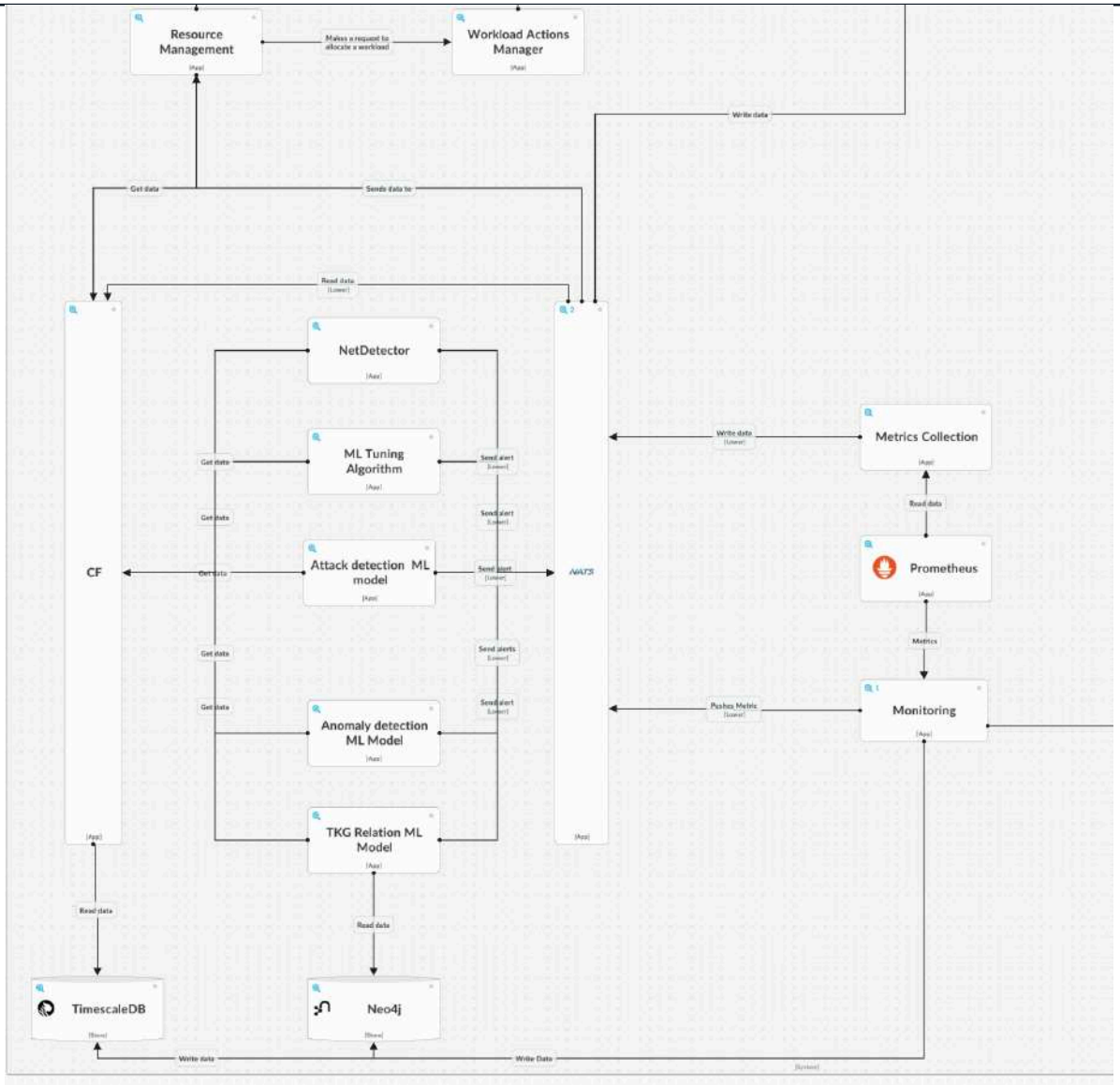

Figure 18: CF relations

As it could be observed from Figure 18, the CF is responsible for providing the read/write access for the components associated with ML predictions. CF is integrated with Net Detector, ML Tuning Algorithm, Attack Detection, Anomaly Detection and TKG relation models. All of them are being deployed as separate services. On the other side CF provides data for Resource Management service developed by Lake and having scheduler inside. To simplify data access, CF provides a Rest API that allows read and write operations for all the available data resources. The full list of requests may be observed on Figure 19.

Figure 19: CF API

With data being the first vital component in the ML process of decision making, the second important piece of the puzzle are models themselves. Keeping this in mind CF assists the ACES platform in this aspect as well. This involves the functionality of providing preconfigured ML models that were designed for a particular set of tasks, as well as storing already fine-tuned models developed by partners inside the corresponding service of ACES platform. Those models are being stored in CF and could be access via API as well.

7 Conclusions

This document has presented a comprehensive overview of the strategies and mechanisms developed within the ACES project to address the challenges of distributed knowledge and data management in edge-cloud environments. By integrating temporal knowledge graphs, cognitive agents, decentralized optimization algorithms, and adaptive data management frameworks, ACES enables resilient, autonomous, and scalable edge intelligence. The proposed methods, including self-organizing scheduling, anomaly detection through ML and GNNs, and decentralized hyperparameter optimization, collectively support the realization of autopoietic edge-cloud services. These innovations lay a robust foundation for future developments toward fully autonomous, context-aware, and self-managing distributed systems.

8 References

[1] https://www.etsi.org/committee/cim
[2] https://nats.io/
[3] https://kubernetes.io/
[4] https://github.com/GoogleCloudPlatform/microservices-demo
[5] https://istio.io/
[6] C. Carrion, "Kubernetes scheduling: Taxonomy, ongoing issues and challenges," ACM Computing Surveys, vol. 55, no. 7, pp. 1–37, 2022.
[7] X. Wang, K. Zhao, and B. Qin, "Optimization of task-scheduling strategy in edge kubernetes clusters based on deep reinforcement learning," Mathematics, vol. 11, no. 20, 2023.
[8] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand et al., "Autopilot: workload autoscaling at google," in Proceedings of the Fifteenth European Conference on Computer Systems, 2020, pp. 1–16.
[9] C. Lu, K. Ye, G. Xu, C.-Z. Xu, and T. Bai, "Imbalance in the cloud: An analysis on alibaba cluster trace," in 2017 IEEE International Conference on Big Data (Big Data). IEEE, 2017, pp. 2884–2892.
[10] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao, "Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces," in Proceedings of the international symposium on quality of service, 2019, pp. 1–10.
[11] Z. Zhong, M. Xu, M. A. Rodriguez, C. Xu, and R. Buyya, "Machine learning-based orchestration of containers: A taxonomy and future directions," ACM Computing Surveys (CSUR), vol. 54, no. 10s, pp. 1–35, 2022.
[12] Y. Xie, M. Jin, Z. Zou, G. Xu, D. Feng, W. Liu, and D. Long, "Real-time prediction of docker container resource load based on a hybrid model of arima and triple exponential smoothing," IEEE Transactions on Cloud Computing, vol. 10, no. 2, pp. 1386–1401, 2020.
[13] X. Sun, C. Hu, R. Yang, P. Garraghan, T. Wo, J. Xu, J. Zhu, and C. Li, "Rose: Cluster resource scheduling via speculative over-subscription," in 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS). IEEE, 2018, pp. 949–960.
[14] Eric Brochu, Vlad M Cora, and Nando De Freitas, "A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning," arXiv preprint arXiv:1012.2599, 2010.
[15] C. Malherbe and N. Vayatis, "Global optimization of Lipschitz functions," International Conference on Machine Learning, pp. 2314–2323, 2017.
[16] A. Bemporad, "Global optimization via inverse distance weighting and radial basis functions," Computational Optimization and Applications, vol. 77, pp. 571–595, 2020.
[17] L. Sabug Jr., F. Ruiz, and L. Fagiano, "SMGO: a set membership approach to data-driven global optimization," Automatica, vol. 133, pp.109890, 2021.
[18] G. Carnevale, F. Farina, I. Notarnicola, and G. Notarstefano, "Distributed online optimization via gradient tracking with adaptive momentum," arXiv:2009.01745, 2020.
[19] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," arXiv:1412.6980, 2014.
[20] https://github.com/ACES-EU
[21] F. Farina, A. Camisa, A. Testa, I. Notarnicola, and G. Notarstefano, "Disropt: a python framework for distributed optimization," IFACPapersOnLine, vol. 53, no. 2, pp. 2666–2671, 2020.
[22] L. Xiao, S. Boyd, and S. Lall, "A scheme for robust distributed sensor fusion based on average consensus," Fourth Internetional Symposium on Information Processing in Sensor Networks, pp. 63–70, 2005.
[23] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[24] https://ozone.apache.org/
[25] https://min.io/
[26] https://ipfs.tech/
[27] https://opentelemetry.io/
[28] https://www.timescale.com/
[29] https://neo4j.com/

[30] Series of blog posts on data management with Nuvla: https://sixsq.com/blog/tech-corner/2020/05/04/data-management-with-nuvla-part-1.html, https://sixsq.com/blog/tech-corner/2020/06/05/data-management-with-nuvla-part-2.html, https://sixsq.com/blog/tech-corner/2020/08/11/data-management-with-nuvla-part-3.html