TRUSTWORTHY, COGNITIVE AND AI-DRIVEN COLLABORATIVE ASSOCIATIONS OF IOT DEVICES AND EDGE RESOURCES FOR DATA PROCESSING

Grant Agreement no. 101136024

Deliverable D4.2
Intelligent Resource Management, Cyber Threat Intelligence and EMPYREAN Aggregator

Programme:	HORIZON-CL4-2023-DATA-01-04
Project number:	101136024
Project acronym:	EMPYREAN
Start/End date:	01/02/2024 – 31/01/2027

Deliverable type:	Report
Related WP:	WP4
Responsible Editor:	ICCS
Due date:	30/04/2025
Actual submission date:	30/04/2025

Dissemination level:	Public
Revision:	FINAL

This project has received funding from the European Union's Horizon Europe research and innovation programme under grant agreement No 101136024

Revision History

Date	Editor	Status	Version	Changes
04.02.25	ICCS	Draft	0.1	Initial ToC
17.03.25	ICCS	Draft	0.2	Integrate initial contributions from NEC, UMU, ICSS in Sections 5,6,7, 9
27.03.25	ICCS	Draft	0.3	Integrate contributions from ICCS, RYAX, NEC, UMU, NUBIS in Sections 6,7,8,9,10,11
04.04.25	ICCS	Draft	0.4	Integrate final contributions in Sections 1,2,3,4,12
09.04.25	ICCS	Draft	0.5	Integrate updates from UMU, ICCS in Sections 9, 11
11.04.25	ICCS	Draft	0.6	Complete version for internal review
25.04.25	ICCS	Draft	0.7	Updates and revisions based on internal review feedback
28.04.25	ICCS	Final	1.0	

Author List

Organization	Authors
ICCS	Aristotelis Kretsis, Panagiotis Kokkinos, Fotis Kouzinos, Nikos Kapentzonis, Kwstas Stergiotis, Emmanouel Varvarigos
NEC	Jaime Fúster, Roberto González
RYAX	Yiannis Georgiou
UMU	Antonio Skarmeta, Eduardo Canovas, Alonso Sanchez
NUBIS	Anastasios Nanos

Internal Reviewers

Alonso Sanchez (UMU)

Ivan Paez (ZSCALE)

Abstract: Deliverable D4.2 presents the key outcomes of the activities that took place in the context of Task 4.1 *"Cyber Threat Intelligence, Intelligent Resource Management, and Energy Efficiency"* and Task 4.4 *"EMPYREAN Aggregator, Autonomous Management and Monitoring"* during the first iteration of the incremental implementation plan (M04-M15). These tasks focus on the design and development of critical components within the EMPYREAN platform, including: (i) cyber threat intelligence through advanced mechanisms for detecting, analyzing, and mitigating cybersecurity threats within the EMPYREAN platform; (ii) intelligent resource management algorithms for optimizing resource allocation across heterogeneous edge-cloud environments, improving efficiency, performance, and resilience; (iii) the EMPYREAN Aggregator, a framework enabling autonomous management, monitoring, and data-driven decision-making within and across the EMPYREAN Associations; and (iv) the EMPYREAN Registry, a unified system for registering, cataloging, and managing resources, services, and Associations within the EMPYREAN ecosystem.

The deliverable details the methodologies, architectural designs, and initial implementation results achieved in these domains. It lays a solid foundation for further advancements and refinements in subsequent iterations, driving the evolution of EMPYREAN's cyber-resilient, intelligent, and self-adaptive computing infrastructure.

Keywords: Edge Cloud Continuum, EMPYREAN Platform, EMPYREAN Components, Associations, Cognitive Orchestration, Cyber Threat Intelligence, Resource Orchestration, Multi-Objective Optimization, Telemetry Service

Table of Contents

List of Figures

List of Tables

Abbreviations

AI	Artificial Intelligence
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
ARIMA	Auto Regressive Integrated Moving Average
ASN	Autonomous System Numbers
ATR	Autonomous Towing Robot
C.U.	Cost Unit
C2C	Command-to-Command
CDN	Content Delivery Network
CNCF	Cloud Native Computing Foundation
CTA	Cyber Threat Alliance
CTC	Cyber Threat Coalition
CTI	Cyber Threat Intelligence
CVE	Common Vulnerabilities and Exposures
D	Deliverable
DAG	Directed Acyclic Graph
DDPG	Deep Deterministic Policy Gradient
DDQN	Double Deep Q-Network
DDR	Double-Data-Read
DE	Decision Engine
DKG	Distributed Knowledge Graph
DRL	Deep Reinforcement Learning
DVFS	Dynamic Voltage Frequency Scaling
FKG	Federated Knowledge Graph
FL	Federated Learning
FVG	Feature Value Grouping
GBM	Geometric Brownian Motion
ILP	Integer Liner Programming
IoC	Indicators of Compromise
IODEF	Incident Object Description Exchange Format
IoT	Internet of Things
JSON	JavaScript Object Notation
K3s	Lightweight Kubernetes
K8s	Kubernetes
KG	Knowledge Graph
KPI	Key Performance Indicator
L.U.	Latency Unit
LSTM	Long Short-Term Memory Network
M	Month
MDN	Malware Distribution Network
MILP	Mixed Integer Linear Programming
MIP	Mixed Integer Programming
MISP	Malware Information Sharing Platform
ML	Machine Learning
NBI	North Bound Interfaces

OCI	Open Container Initiative
OF	Operation Flow
OTEL	Open Telemetry
PHA	Potentially Harmful Applications
PMDS	Persistent Monitoring Data Storage
PPFL	Privacy Preserving Federated Learning
PSM	Privacy and Security Manager
PUP	Potential Unwanted Program
QoS	Quality of Service
RDF	Resource Description Framework
REST	Representational State Transfer
ROT	Resource Optimization Toolkit
RRN	Recurrent Neural Network
SDO	STIX Domain Object
SIEM	Security Information and Event Management
SLO	Service Level Objectives
SotA	State-of-the-Art
SRA	Speculative Resource Allocation
SRO	STIX Relationship Object
STIX	Structured Threat Information Expression
SVM	Support Vector Machine
TAXII	Trusted Automated Exchange of Intelligence Information
UAV	Unmanned Aerial Vehicle
UC	Use Case
UCO	Unified Cybersecurity Ontology
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
VM	Virtual Machine
VMI	Virtual Machine Introspection
WASM	Web Assembly
WP	Work Package
XR	Extended Reality

1 Executive Summary

EMPYREAN seamlessly integrates IoT devices, robotic systems, and computational resources into collaborative, dynamic collectives, the Associations. This Association-based continuum forms an autonomous and interconnected ecosystem designed to support hyper-distributed applications. At the core of EMPYREAN's platform lies an AI-enabled control and management plane, which enables efficient and adaptive operations by optimizing resource utilization, system performance, and resiliency across Associations.

Deliverable 4.2 documents the progress achieved during the first implementation cycle (M4-M15), specifically focusing on the work carried out in two of the four tasks within WP4. These tasks are dedicated to the implementation of intelligent resource management algorithms, the development of novel cyber threat intelligence mechanisms, along with the design and development of core orchestration and management services.

The document is organized into nine key chapters, each detailing a major building block of the EMPYREAN framework. Section 3 outlines the overall architecture and highlights the components relevant to this deliverable. Section 4 describes how these components interoperate to support cross-layer orchestration, intelligent management, cyber threat intelligence, and distributed coordination. Section 5 details the strategies and mechanisms developed for intelligent resource allocation and dynamic workload balancing across heterogeneous infrastructures. Section 6 presents the development of the Decision Engine, which provides decision-making capabilities within the EMPYREAN platform. Section 7 elaborates on the cyber threat intelligence capabilities developed to ensure resilience and trustworthiness, focusing on intrusion detection, behavioral analysis, and automated response mechanisms. Section 8 describes the Service Orchestrator, which governs service lifecycle management, deployment placement, and elastic adaptation in Associations. Section 9 presents the Telemetry Service, responsible for collecting, filtering, and processing performance and operational data to support both real-time responsiveness and long-term system optimization. Section 10 and 11 cover the EMPYREAN Aggregator and EMPYREAN Registry, two critical components of the control and management plane. They manage the operations of the Association-based continuum, maintain system-wide awareness, and enable secure, scalable, and transparent service discovery and deployment.

The developments described in this deliverable are key contributors to the initial release of the EMPYREAN platform. Furthermore, the developed mechanisms play a significant role in supporting the initial implementations of EMPYREAN's use cases, which will be detailed in D5.1 "Use cases technological developments" (M18). As the project advances, these foundational developments will be iteratively refined and based on feedback gained through ongoing integration activities. The final version of these mechanisms will be presented in D4.3 "Final report on decentralized intelligence, application development and deployment" (M26).

2 Introduction

2.1 Purpose of this document

Deliverable D4.2 presents the outcomes of Task 4.1 *"Cyber Threat Intelligence, Intelligent Resource Management and Energy Efficiency"* and Task 4.4 *"EMPYREAN Aggregator, Autonomous Management and Monitoring Fabric"*, as part of the first iteration of the incremental implementation plan (M04-M15). T4.1 relates to the development of novel multi-agent and multi-objective algorithms for resource allocation and service orchestration, alongside advanced cyber threat intelligence mechanisms. These solutions aim to enable collaborative autonomy, cognitive operation, and secure operation within the EMPYREAN platform. T4.4 focuses on developing mechanisms that support the cognitive, cooperative, and autonomous management of the Association-based continuum, also promoting self-driven adaptability.

The objective of D4.2 is to build on the final architecture of the EMPYREAN platform, as outlined in deliverable D2.3 (M12), towards the provision of the initial release of the EMPYREAN orchestration and decision-making mechanisms (i.e., Service Orchestrator, EMPYREAN Controller, Decision Engine), advanced cyber threat intelligence service (i.e., CTI Engine), resource allocation algorithms, association management mechanisms (i.e., EMPYREAN Aggregator, EMPYREAN Registry), and telemetry mechanisms (i.e., Telemetry Engine, Monitoring Probes, Persistent Monitoring Data Storage).

D4.3 "Final report on decentralized intelligence, application development and deployment" in M26 will present the final release of the EMPYREAN components and mechanisms that are developed in the context of T4.1 and T4.4.

2.2 Document Structure

The present deliverable is split into nine major chapters:

- EMPYREAN Architecture Mapping
- EMPYREAN Platform
- Intelligent Resource Management
- Decision Engine
- Cyber Threat Intelligence
- Service Orchestrator
- Telemetry Service
- EMPYREAN Aggregator
- EMPYREAN Registry

2.3 Audience

This document is publicly available and intended for anyone interested in the initial description of the EMPYREAN mechanisms related to cyber threat intelligence, intelligent workload allocation and resource management, telemetry service, autonomous management of EMPYREAN Associations, and service orchestration and application deployment mechanisms within and across Associations. Additionally, it serves as a valuable resource for the general public, providing insights into the design and implementation of the above core mechanisms, as well as their role in addressing the requirements of the project's use cases.

3 EMPYREAN Architecture Mapping

The EMPYREAN architecture was first introduced in deliverable D2.2 "Initial Release of EMPYREAN Architecture" (M07), and later refined in its final version in D2.3 " Final EMPYREAN architecture, use cases analysis and KPIs" (M12). This refinement incorporated key insights gained from the initial implementation phase. D2.3 provides a comprehensive overview of the architecture, detailing the EMPYREAN components, their interfaces, and the supported operational flows.

In this section, we present a concise description of the architecture (Figure 1) to support the discussion of the initial developments in WP4, particularly focusing on cyber threat intelligence, multi-objective resource allocation and service orchestration algorithms, and distributed and autonomous Association management and telemetry mechanisms within the EMPYREAN.

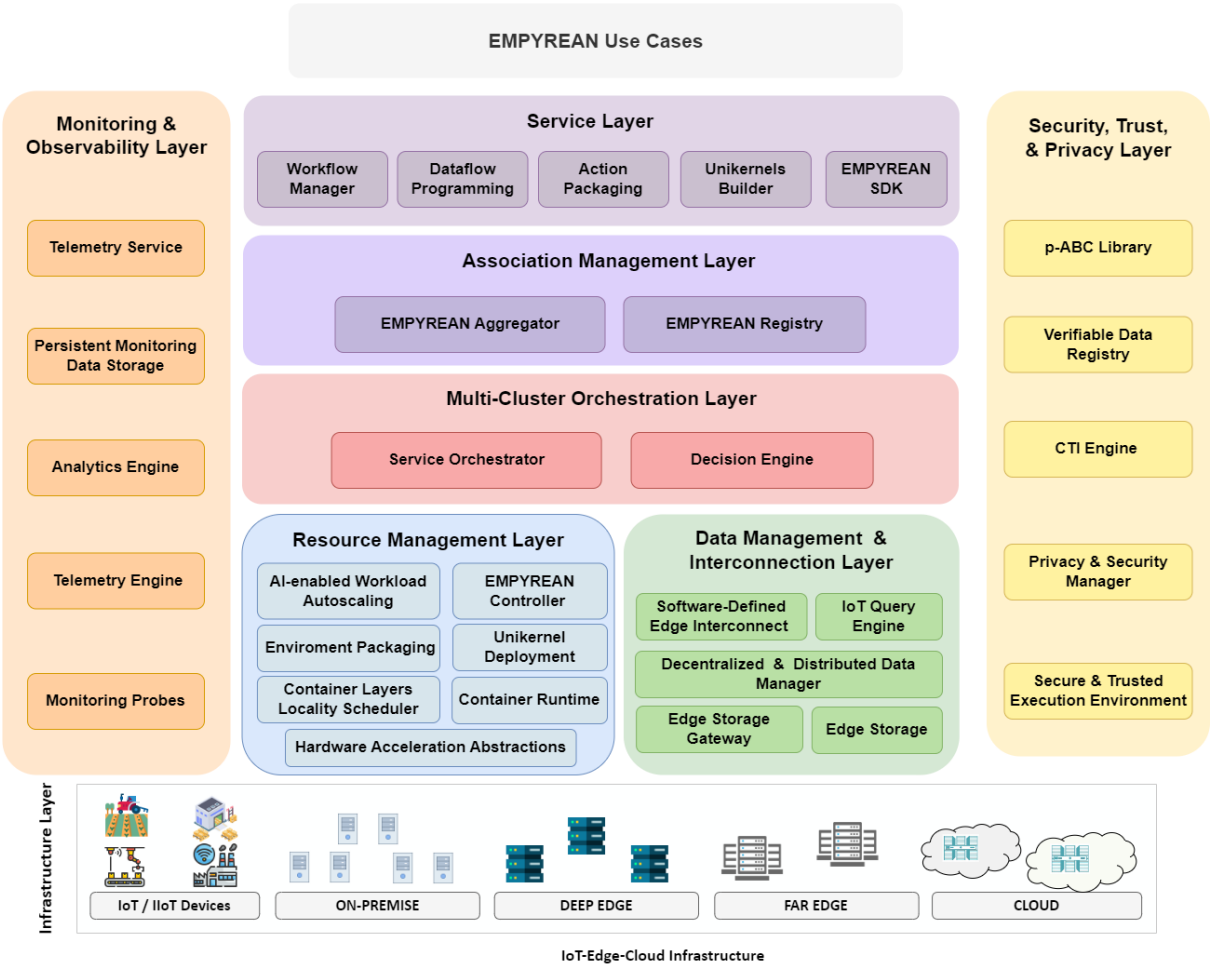


Figure 1: EMPYREAN high-level architecture.

The ***Service*** layer facilitates the development of Association-native applications, providing robust support for application-level adaptations, interoperability, elasticity, and scalability across the IoT-edge-cloud continuum. Deliverable D4.1 (M15) provides a detailed description for the design and development of this layer's components.

The ***Association Management*** layer includes components that intelligently and dynamically create and manage Associations. Each Association integrates heterogeneous resources across multiple providers, connectivity types, and segments of the IoT-edge-cloud continuum. The *EMPYREAN Registry* (Section 11) manages the registration of IoT devices, edge, and cloud resources within Associations while tracking available services and resources. It offers an abstract and unified view of the Association-based continuum, simplifying application development and deployment. Moreover, the *EMPYREAN Aggregator* (Section 10) handles the formation, coordination, and management of Associations, while it facilitates the discovery of available resources. Multiple self-managed and interacting Aggregators constitute the distributed and data-driven management plane for the EMPYREAN platform.

The ***Multi-Cluster Orchestration*** layer enables efficient service orchestration and resource management across the disaggregated and heterogeneous EMPYREAN infrastructure. Through autonomous, distributed decision-making mechanisms, this layer orchestrates dynamic, hyper-distributed applications while enabling self-driven adaptations. Multiple instances of its components provide decentralized operation, optimizing resource utilization while ensuring scalability, resiliency, energy efficiency, and service quality.

The *Service Orchestrator* (Section 8) oversees application deployment and coordinates the necessary resource management actions. Workload distribution and assignment decisions are delegated to the *Decision Engine* (Section 6), which enables decentralized, speculative, and multi-objective resource orchestration. The Decision Engine integrates various distributed optimization and orchestration algorithms (Section 5) to balance computing tasks and data both locally within an Association and across federated Associations.

The ***Resource Management*** layer unifies IoT, edge, and cloud platform management within the EMPYREAN platform. Operating within Kubernetes (K8s) or lightweight Kubernetes (K3s) clusters, this layer ensures modularity, facilitating seamless hardware and software integration.

The *EMPYREAN Controller (*Section 8*)* serves as a bridge, integrating individual IoT, edge, and cloud resources under the control of a specific Service Orchestrator. The *AI-enabled Workload Autoscaling* component, detailed in deliverable D3.2 (M15), enhances Kubernetes orchestration capabilities by incorporating AI/ML techniques for intelligent workload autoscaling. The *Environment Packaging* component, deliverable D4.1 (M15), supports multi-environment and multi-architecture packaging for cloud-native applications, improving the interoperability and adaptability of workloads. The *Unikernel Deployment and Container Runtime* components, presented in D4.1 (M15), enable flexible container runtime integration, allowing cloud-native applications to deploy across various execution environments. The *Container Layers Locality Scheduler,* implemented as a scheduling plugin for the local orchestrator in each platform, optimizes workload scheduling at the cluster level. Finally, the

Hardware Acceleration Abstractions component, deliverable D3.2 (M15), enables the offloading of compute-intensive tasks to hardware accelerators on neighbouring nodes.

The ***Data Management and Interconnection*** layer implements dynamic communication and secure data storage across IoT devices and computing resources. Operating at both the cluster and Association levels, this layer provides flexible and scalable data management while seamlessly integrating IoT, edge, and cloud resources. Deliverables D3.2 (M15) and D4.1 (M15) provide further details on the developments of its components.

Across the EMPYREAN ecosystem resides the ***Monitoring and Observability*** layer that autonomously collects and analyses telemetry data across the Association-based infrastructure and deployed hyper-distributed applications. The EMPYREAN Telemetry Service (Section 9) consists of three key components: the *Telemetry Engine, Monitoring Probes, and Persistent Monitoring Data* Storage. These components enable the data collection, preprocessing, correlation, and management, facilitating orchestration and service assurance mechanisms to optimize performance and reliability. Complementing this, the Analytics Engine, detailed in deliverable D3.2 (M15), forms a crucial part of this layer by enabling advanced data analysis and insights.

The ***Security, Trust, and Privacy*** layer ensures secure access, data privacy, and trusted execution across the EMPYREAN platform, spanning both cluster and Association levels. The *CTI Engine* (Section 7) delivers automated cyber threat analysis, providing valuable intelligence on past global cyber threats. By quantifying system risks, the CTI Engine enables proactive security adaptations within and across EMPYREAN Associations, significantly strengthening the platform's overall security capabilities.

4 EMPYREAN Platform

The EMPYREAN platform (Figure 2) is designed as a self-optimizing system that continuously adapts to its environment by executing a cognitive cycle of sensing (detecting system and environmental changes), discerning (interpreting senses), inferring (understand implications), deciding (selecting appropriate actions), and acting (executing decisions). This cognitive intelligence loop is foundational to EMPYREAN's ability to support robust, autonomous operations across the IoT-edge-cloud continuum.

Key components such as the Cyber Threat Intelligence (CTI) framework, Association management and coordination mechanisms, and intelligent service and resource orchestration collectively provide an abstraction layer that automates platform operations while optimizing the utilization of heterogeneous, distributed resources. These capabilities underpin the Association-based continuum, enabling the seamless deployment and coordination of hyper-distributed, cloud-native applications spanning IoT, edge, and cloud infrastructures.

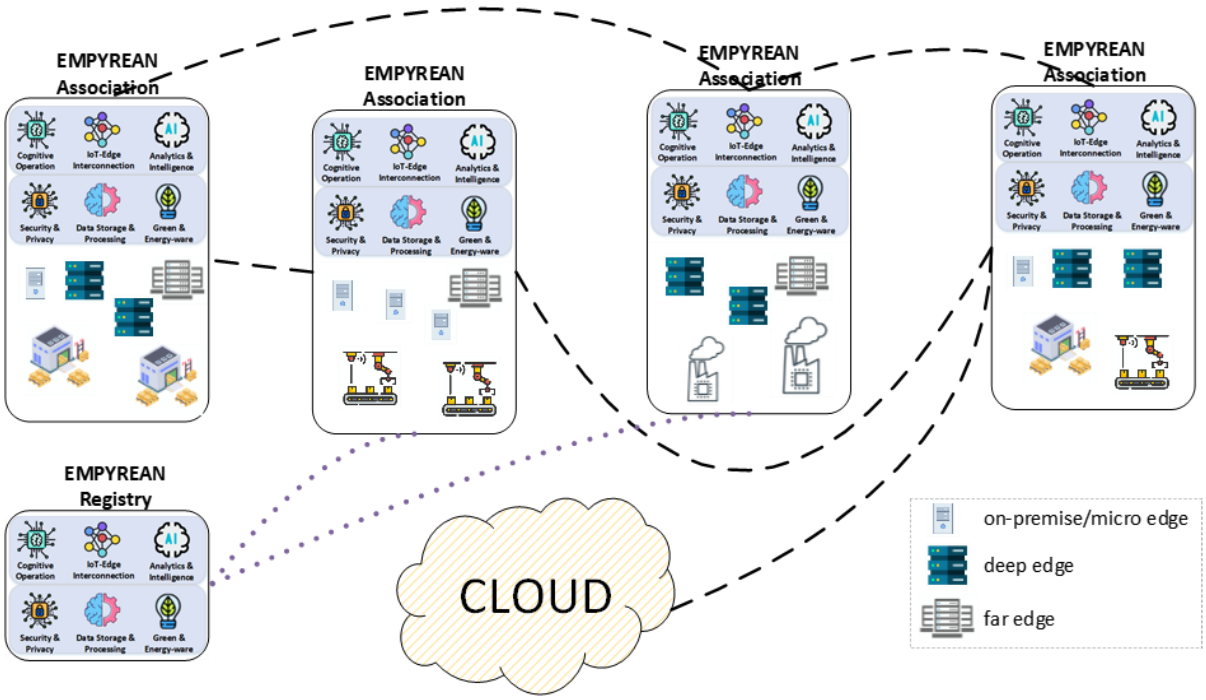


Figure 2: EMPYREAN Association-based IoT-Edge-Cloud continuum.

The EMPYREAN Registry and EMPYREAN Aggregator are key enablers of autonomous collaboration, facilitating efficient deployment, dynamic management, and coordinated operation of both Associations and their hosted applications. The architecture includes one EMPYREAN Registry and multiple Aggregators for the management of the available Associations. As core components of the control and management plane, they provide administrators and authorized infrastructure providers with full Association lifecycle management capabilities, including the ability to create, update, and delete Associations.

Throughout these operations, the Registry and Aggregators interact with the platform's security plane components, such as the Privacy and Security Manager and Distributed Ledger Technology, as well as the Telemetry Service, ensuring that all changes are trusted and aligned with the system's governance policies. The interaction flows corresponding to these procedures are described in detail in D2.3 (M12) under Operation Flows (OF) 1.1, 1.2, and 1.3.

Following the setup of Associations, which create secure and collaborative execution environments, the next operational phases include the resource onboarding and application development. During these stages, the EMPYREAN Registry, along with multiple Aggregator instances, play a critical role in the registration and integration of resources. These components interface primarily with the Resource Management and the Security, Trust, and Privacy layers to onboard heterogeneous resources, register their capabilities and constraints, and define access control policies and operational processes that govern their participation in the Associations.

Next, the application deployment is structured into three coordinated phases: (i) high-level assignment of cloud-native application microservices to relevant EMPYREAN Associations (OF4.1.1); (ii) cognitive orchestration of assigned microservices within the selected Associations (OF4.1.2); and (iii) seamless deployment of workloads across the selected Kubernetes (K8s) or lightweight Kubernetes (K3s) clusters (OF4.1.3).

In this context, the Association Management layer components, comprising the EMPYREAN Registry and Aggregator, is tightly integrated with the Multi-Cluster Orchestration layer, which includes the Service Orchestrator and Decision Engine. These components collaborate during the first two phases to enable intelligent, constraint-aware resource orchestration across the EMPYREAN continuum. The orchestration mechanisms consider a wide range of key criteria, including latency requirements, performance objectives, energy efficiency, and security requirements, ensuring that workloads are optimally distributed to suitable Associations, clusters, and infrastructure resources.

In the final phase, the Service Orchestrator interfaces with components from the Resource Management layer, such as the EMPYREAN Controller and the Containers Layer Locality Scheduler, to execute the actual workload placement and deployment. These components are responsible for selecting the most suitable worker nodes within the targeted K8s/K3s clusters and for carrying out the deployment operations based on declarative specifications defined in the earlier phases.

In addition, to supporting collaborative and distributed operations, the EMPYREAN platform automates system-level operations to achieve continuous optimizations and self-management. Central to this capability is the Telemetry Service, which provides a distributed, real-time monitoring infrastructure that supports the comprehensive observation of system behavior. It enables continuous data collection and the generation of actionable insights to optimize both the platform's performance and that of deployed applications. The service monitors a wide range of entities, including infrastructure resources, robots, IoT devices, and running workloads. In addition, it captures energy consumption metrics across all resource types, contributing to sustainability and energy efficiency goals. The collected telemetry data

serves as a vital input for EMPYREAN's distributed decision-making and analytics mechanisms, feeding into intelligent policies for dynamic optimization, anomaly detection, and resource-aware workload management.

Complementing this, the Cyber Threat Intelligence (CTI) Engine collects and analyses data from trusted sources such as the Cyber Threat Alliance (CTA) and Malware Information Sharing Platform (MISP) repositories. By extracting relevant patterns and critical indicators, the CTI Engine enhances the platform's situational awareness and proactive defense capabilities. The CTI framework is integrated with the Telemetry Service, allowing it to ingest real-time monitoring data from across the entire EMPYREAN platform. It also interfaces with Association-level orchestration and analysis tools, such as the EMPYREAN Aggregator and Analytics Engine, to enable localized threat detection, context-aware analysis, and timely response actions. This integration supports automated threat intelligence workflows, enabling seamless security analytics and dynamic workload and data migration in response to emerging risks across Associations.

5 Intelligent Resource Management

EMPYREAN embraces a distributed, speculative, and intelligent approach to orchestrating hyper-distributed applications, striking a dynamic balance between centralized and decentralized paradigms. This approach enables a more adaptable and efficient computational continuum, ensuring that workload distribution and resource management are continuously optimized based on real-time conditions, performance constraints, and sustainability goals.

To achieve this, EMPYREAN is developing a suite of novel algorithms that leverage multi-objective optimization, game theory, AI/ML techniques, and heuristic methods. These algorithms are designed to balance trade-offs between optimality and computational complexity, ensuring that the system remains scalable, responsive, and efficient even as workloads and environmental conditions evolve.

5.1 Optimization of Cloud-Native Application Execution over the Edge-Cloud continuum enabled by DVFS

5.1.1 Introduction

The increasing complexity of modern applications is progressively pushing traditional monolithic designs out of the spotlight. As a result, the cloud native application model [2] is adopted, leading to a paradigm shift from classic monolithic structures to flexible microservice-based architectures. This trend is evident in a wide range of new-era, AI-enhanced applications: industrial control, video analytics, interactive (XR) media, remote healthcare, autonomous vehicles, smart agriculture, and general smart city services, among others. In the context of EMPYREAN, both the generic IoT applications and the AI-powered applications of end-users inside associations can be considered microservice-based. The fine-grained decomposition of monolithic entities into distinct components grants significant advantages in terms of performance, scalability, and flexibility. Deploying and scaling each component individually enables the application to span multi-tenancy and multi-technology environments in search for the most suitable resource type according to its specific requirements. This is in accordance with the hyper-distributed resource hierarchy of EMPYREAN, spanning from low-power edge devices to high-end cloud computers within resource Associations.

However, with the emergence of new applications, their requirements become more stringent, especially in terms of end-to-end latency, creating challenges for the resource orchestration mechanisms. For instance, remote surgery operations require immediate responses, even below 1 millisecond [10]. From the infrastructure operator's perspective, increased latency can result in revenue loss, with Amazon reporting that every additional 100 ms of experienced user latency incurs a 1% loss during traffic spikes [7].

To address the limitations of the traditional "all-to-cloud" approach- in terms of delay when transmitting to remote data-centers- computation resources are deployed at the network periphery, giving rise to the concept of edge computing. This architecture model complements cloud computing by alleviating part of the computing burden, while minimizing latency thanks to the spatial proximity of the edge servers to data-generation points [1]. Unfortunately, the edge layer inherently possesses a mere fraction of the cloud's computing capacity, while the associated hardware is generally less powerful in terms of performance [5].

Microservices, despite being dedicated and loosely coupled, they are not entirely self-contained in practice. Communication-based dependencies manifest in various forms, such as data exchange, querying, and result forwarding flows [13], forming complex execution paths. These paths emerge at runtime and can be accurately represented by Directed Acyclic Graphs (DAGs) [4], where each node represents a microservice and each arc signifies a downstream relationship between connected services. The critical path is the path with the longest end-to-end latency, thus bounding the total execution time of the application.

Accelerating the processing of a service can be achieved through various methods. Code-level optimizations—such as vectorization, parallelization, and improved data locality—can significantly boost performance but rely heavily on developer expertise. From the resource orchestrator perspective, services can be accelerated by deploying them on high-performance systems, such as a cloud CPUs (e.g., Intel Xeon Gold), GPUs (e.g., NVIDIA RTX 4070) or even hardware accelerators (e.g., Iceriver KS5L ASIC). On a more fine-grained level, this work also explores the acceleration of services by increasing the processing unit's frequency. To this end, we leverage Dynamic Voltage Frequency Scaling (DVFS), a well-established technique that allows processors to adjust their operating frequency and voltage, exploiting a trade-off between performance and energy efficiency [16] during the resource allocation process. Strategically applying DVFS based on the criticality of services within the application's DAG enables performance gains while minimizing power wastage. The latter is of paramount importance, not only for its environmental impact but also for the sustainability and longevity of the infrastructure.

The aim of this work is to optimize microservice-based applications on the Association edge-cloud continuum by assigning microservices to infrastructure nodes and processing devices. We only consider the internal hierarchical resource-allocation within one association, with the high-level decisions guided from the EMPYREAN Decision Engine and subsequently offloaded to local orchestrators. Leveraging the DAG structure of the applications, we can identify critical and non-critical execution paths. This allows us to accelerate critical services—by placing them on high-end processors operating at higher frequencies—to minimize overall execution time, while conserving power and energy in less congested areas—by deploying services on low-power, lower-frequency devices. By taking advantage of the highly heterogenous resource pool of EMPYREAN Associations and by strategically fine-tuning the frequency of the resources, we show that it is possible to optimize performance without excessive energy consumption or allow "green" execution without significantly compromising execution time.

5.1.2 Related Work

Processor frequency scaling has been extensively studied in contemporary research to manage the performance-energy consumption trade-off. In [23], the authors employ DVFS to optimize energy consumption in ultra-low-power embedded systems. They propose a mechanism that dynamically adjusts the processor's frequency, operating at the lowest frequency during periods of low demand and scaling up during intensive tasks to maximize efficiency. Similarly, the study in [6] investigates the trade-off between performance and energy consumption by utilizing DVFS and thread scaling on server processors. The results highlight that higher frequencies yield performance gains; interestingly, energy efficiency is not attended at the minimum frequency, as a consequence of the prolonged execution time.

Garcia et al. [8] analyze the impact of various policies implemented by Linux Governors (e.g., *performance, powersave, ondemand*) on performance and energy efficiency. Additionally, in [16], different voltage and frequency settings combined with various process placement strategies are explored to assess their effects on performance and energy efficiency.

Application acceleration can be enhanced by targeting specific bottlenecked services instead of dealing with the entirety of a microservice-based application. CRISP [22] employs critical path analysis over large traces of microservice call graphs in order to pinpoint and optimize crucial services. The mechanism was deployed across the entire UBER network and successfully identified optimization opportunities. The authors of [17] introduce FIRM, a ML-enabled framework aiming to reduce service level objective (SLO) violations in microservice-based application workloads. A support vector machine (SVM) mechanism is employed to initially detect critical paths in application structures and subsequently single out the specific services responsible for SLO violations. A Deep Deterministic Policy Gradient (DDPG) reinforcement learning algorithm is developed to efficiently re-provision resources on the critical services.

Somashekar et al. [18] investigate the problem of fine-tuning individual configuration parameters for microservices that lie on the critical path of application structures. A dimensionality reduction technique is utilized to reduce the exploration space by identifying only a subset that contains the most important configuration parameters for each service. Then, the fine-tuning is performed at runtime by an iterative process that perturbs the existing configuration and evaluates the result. Song et al. [19] demonstrate ChainsFormer, a framework that identifies critical chains and nodes in microservice-based applications based on a predictor module, and subsequently provision resources leveraging a SARSA reinforcement learning algorithm.

Previous studies generally focus on specific resources or clusters (e.g., a single data center). In contrast, our study examines the complexities of serving microservice-based applications over the edge-cloud continuum, accounting for device heterogeneity and communication delays. While prior work primarily enhances resources through traditional horizontal (replication) and vertical (resource augmentation) scaling, we leverage, for the first time, DVFS along with the application’s dependencies, represented as a DAG, to determine optimal configurations for microservices. Additionally, related studies often assume a fixed deployment scheme and only

enhance the pre-established critical path at runtime, which can lead to inefficient resource use as new critical paths and services emerge. To our knowledge, though this limitation is recognized in the literature, it has not been directly addressed. Our approach, however, considers all possible execution paths of the application during runtime and attempts to identify the optimal configuration based on the resulting critical path.

5.1.3 System Model

5.1.3.1 DVFS, execution time and power consumption model

Dynamic Voltage-Frequency Scaling enables the dynamic adjustment of the frequency at which a processing unit operates, based on the current workload and desired objectives (e.g., minimizing power consumption or maximizing performance). By increasing the CPU frequency—and consequently the voltage—performance is enhanced during intensive task execution. Conversely, decreasing the CPU frequency conserves power and reduces thermal output, albeit at the expense of reduced performance. The underlying hardware must support multiple power and performance states (often termed P-states), which is common in most modern processors, ranging from edge devices (microprocessors and typical desktop/laptop CPUs) to high-end server processors.

DVFS can be realized through a variety of techniques that provide interfaces, policies, and/or controls to adjust CPU frequencies and voltages at different levels: Linux kernels support frequency scaling configured directly in the OS through the use of governors, which are a set of different policies (e.g., Performance, Power-Save, On-Demand) that automate the scaling process based on the system load and the desired objective. Third party tools such as AMD Ryzen Master and Intel XTU allow for manually scheduling the desired CPU frequency while also providing a set of telemetry tools for monitoring real-time power consumption and component temperature, among others. Moreover, many modern computing systems allow frequency and voltage adjustments directly via the BIOS/UEFI firmware settings.

In this work, we assume the availability of per-core DVFS, allowing the individual and independent tuning of each core in multi-core systems. This fine-grained control adapts to the specific needs of each processing core, aligning with the demands of microservice-based applications. Per-core DVFS is available on most newer-generation processors.

To estimate the execution time of a microservice, which is ultimately a piece of executable code, we employ the well-known formula:

$$T = \frac{\#Instructions \times CPI}{f} \quad (1)$$

In this equation, the number of instructions ($\#Instructions$) can be determined by analyzing the code, while CPI (Cycles Per Instruction) refers to the average number of CPU cycles required to execute one instruction. Different instruction types require varying numbers of cycles. For instance, a typical register bit-wise addition in Assembly requires between one and two cycles, whereas a division operation takes up to 20 cycles. Control operations such as branches (e.g., if statements) and loops heavily depend on the branch prediction mechanism

and can consume several hundred cycles in case of mispredictions. Finally, f represents the operating frequency of the processing unit. This work focuses on compute intensive services, therefore we consider computing frequency to be the determining factor of the execution time. Nevertheless, we can safely assume timely data-fetching using in-memory computations with the newest generation of Double-Data-Read (DDR) RAM memories throughout the infrastructure.

Regarding power consumption (P) and its relationship with frequency, we use the formula:

$$P = C \times V^2 \times f \quad (2)$$

Here C is the effective switching capacitance depending on the chip architecture and activity factor, V is the supply voltage, and f is the operating frequency. At first glance, this formula suggests that power consumption is linearly correlated with f. However, in practice, utilizing DVFS involves scaling up the frequency, which often necessitates a corresponding increase in voltage to maintain stable operation.

The relationship between voltage and frequency is not strictly linear and varies even among processors of the same family due to manufacturing variations—a phenomenon known as the "silicon lottery." Therefore, one can estimate the power at a specific frequency based on surrogate functions such as the one employed in [11] , or any data measured or disclosed by manufacturers or on internal testbeds.

5.1.3.2 Infrastructure description

We consider a hierarchical edge-cloud infrastructure, represented by a graph $G = (V, E)$. Each node $v \in V$ represents a geographical location with collocated devices (e.g., a micro-datacenter) and arcs $e \in E$ describe the networking connections between different nodes. Let D be the total number of distinct types of devices encompassed in the infrastructure, indexed by $d = 1, \dots, D$. A *device* denotes a specific model of a computing system that can execute microservices, ranging from general-purpose consumer CPUs to high-end server processors and GPUs. Each device d possesses a total of C_d processing cores. Therefore, each node $v \in V$ is characterized by a tuple $c_v = [\, c_{v,1}, c_{v,2}, \dots, c_{v,D}]$, indicating the cumulative availability (in terms of the number of available cores) of each type of device at that node (with $c_{v,d} = 0$ if device d is not available at node v, or when all the available cores are currently occupied).

Moreover, each device type $d \in \{1, 2, \dots, D\}$ is characterized by its minimum and maximum operating frequencies f_{min}^d and f_{max}^d, depending on its specifications. We consider a frequency step Δf^d for each type of device d. Hence, the DVFS controller can fine-tune the frequency of a core of a processor d at any level $f^d = f_{min}^d + \alpha \cdot \Delta f^d, a \in N^+$ in the feasible region $F^d = [f_{min}^d, f_{max}^d]$. Each individual frequency level f^d has an associated power consumption $P(f^d)$, based on Equation (2) or any other data disclosed from the manufacturer. Finally, each node $v \in V$ has a "power budget" e_v, which is essentially the maximum power it can sustain at any given time, constrained by the established power policies and cooling systems.

5.1.3.3 Application description

A cloud-native application a encompasses a total of $I_a$microservices, which can be represented by a Directed Acyclic Graph (DAG) $G^a = (V^a, E^a)$. Nodes $v_i \in V^a$ represent the microservices of application a, where $i = 1, 2, \dots, I_a$. Each microservice i of application a is characterized by the tuple $[\, r_i, L_i \,]$. Variable r_i represents the estimated required processing cycles for the service's execution, as the product of the instruction set of its underlying code with the estimated cycles per instruction (Equation (1). Weight L_i is the communication delay limit of the service. Assuming that $g_a \in V$ denotes the data-source node of application a, some microservices are responsible for communicating critical results with the end-user or other entities in a timely manner, and therefore require the delay between the data-source node and the service node v to not exceed their predefined maximum, that is $l_{g_a,v} \leq L_i$. We use $l_{g_a,v}$ to describe the delay, abstracting the underlying physical network connection between the application and an infrastructure node into a single value, which can be determined, for instance, by applying shortest path algorithms.

The execution time of microservice i on a core of device d operating at frequency f^d can be calculated using

$$t_{i,d,f^d} = \delta_{i,d} \times \frac{r_i}{f^d} \quad (3)$$

The coefficient $\delta_{i,d}$ accounts for the fact that high-end server processors generally perform better at the same clock speed (frequency) compared to edge-device processors, due to their faster cache memories, better pipeline structures, support for ECC memory, and advanced branch prediction mechanisms, among others. However, the performance deviations are insignificant for typical consumer applications such as gaming, web-browsing, or standard working online tools. Hence, depending on the type of service and the device, the execution time may vary beyond the simplified formula on Equation (1).

Furthermore, according to [3], time sharing the same physical core—even across different hyper-threads—results in significantly lower throughput for compute-intensive applications due to resource contention (e.g., ALUs, caches, pipelines). Therefore, we adopt a one-to-one mapping between services and cores to ensure that the theoretically achievable execution times are efficiently approximated in practice.

We also consider microservices that require multiple processing cores due to their parallel structure. In our context, such services are modeled as multiple parallel single-core microservices equal in number to the cores required by the original service. Nonetheless, microservices are typically designed to complete specific sub-tasks of an application and usually do not require more than one core.

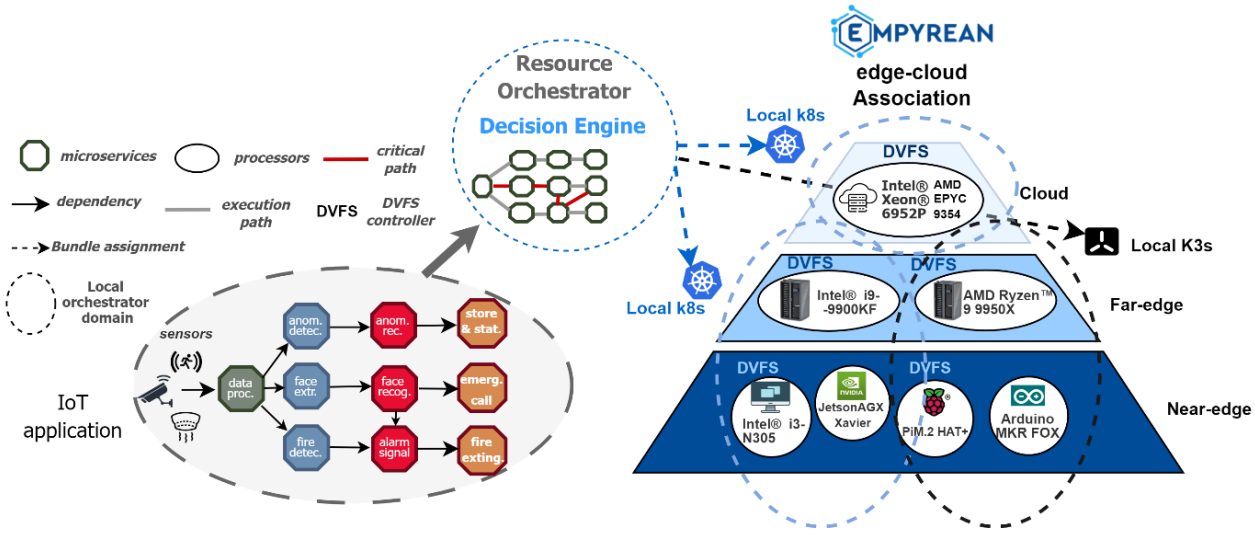


Figure 3: Proposed System Model.

Figure 3 presents an example of a smart surveillance application. Sensor data initiates a sequence of microservice interactions: Smart cameras, smoke and motion detectors are uploading raw streams to a data processing microservice. The latter undertakes data cleaning tasks such as de-colorization, quality enhancement and frame selection. The refined data are forwarded in parallel to three microservices. The first performs anomaly detection, essentially identifying any operational malfunctions of the end devices. If a malfunction is reported, it is forwarded to the subsequent service for further inspection, and finally uploaded for in-depth analysis. The second service performs face extraction on the data, and forwards isolated bounding-boxes to a face recognition microservice, responsible for identifying the depicted people by contrasting the images to a database. The alarm signalling microservice is triggered in case of identification of unauthorized parties, along with the emergency call to the appropriate authority. The fire detection and extinction path are operating accordingly. This application aims to showcase the sequential service invocation that creates different runtime execution paths, as described in Section 5.1.1.

The entire edge-cloud Association is governed by a centralized super-cluster Resource orchestrator in accordance with the EMPYREAN architecture. Upon receiving the application's Directed Acyclic Graph (DAG) and the corresponding microservice requirements, the orchestrator utilizes the proposed mechanisms to determine the optimal configuration for each microservice, specifying the target device, operating frequency, and node assignment. Once the assignments are established, the orchestrator communicates these configurations to the local controllers (local K8S/k3s masters), which then perform the final deployment and frequency tuning, utilizing the per-device DVFS controller agents. The DVFS agents then adjust the frequency of the designated cores on the selected devices accordingly. As discussed earlier, DVFS controllers can range from automated changes in the BIOS/UEFI settings of their assigned devices to OS-level controls and third-party tools that enable frequency scaling.

Our mechanism aims to provide a collective assignment of microservices to infrastructure nodes and devices, with the objective of minimizing a weighted combination of the application's execution time (i.e., the resulting critical path's length) and the total energy consumption, subject to node capacity and power constraints, the delay limits of the microservices, and the available frequency levels of the processing units.

5.1.4 Problem Formulation

Upon selecting an assignment tuple $[node, device, frequency] = [v, d, f]$ for the deployment of each microservice, a set of M paths emerge, each characterized by its execution time $q_m, \forall\, m \in M$. The critical path $\kappa \in M$ is the one with the longest execution time, that is $\$q_\kappa \geq q_m, \forall\, m \in M$. Furthermore, let $m_i = 1\ if\ the\ i^{th}$ microservice is a member of the execution path $m \in M$, and 0 otherwise.

Before we mathematically describe the problem, we introduce some extra notation: Decision variable x_{i,v,d,f^d} equals to 1 if the $\$i^{th}$ microservice is assigned to infrastructure node v and a core of a device of type d, operating at frequency f^d, and 0 otherwise. A tightly coupled variable y_{i,d,f^d} is equal to 1 if there is any node on which $x_{i,v,d,f^d} = 1$. We use y to describe the higher-level assignment on device and frequency level, irregardless of the specific infrastructure node. For ease of reference, the complete set of variables along with the corresponding interpretations are presented in Table \ref{table1}.

The execution time of service i can be defined as:

$$t_i = \sum_{d=1}^{D} \sum_{f^d \in F^d} t_{i,d,f^d} \cdot y_{i,d,f^d} \quad (4).$$

Therefore, the execution time of path $m \in M$, q_m, can be calculated as the cumulative execution time of the services that lie on the path:

$$q_m = \sum_{i=1}^{I_a} t_i \cdot m_i \quad (5).$$

The execution time of the application, T, is the execution time of the resulting critical path:

$$T = q_\kappa = \max_{\mathrm{m}} q_m \quad (6).$$

Table 1: Summary of Notations.

Notation	Interpretation
$G = (V, E)$	Directed weighted graph representing the infrastructure
V	Set of infrastructure nodes
E	Set of network links between nodes
D	Set of different types of processing devices in the infrastructure
C_d	Number of cores of device of type d
$c_{v,d}$	Remaining cumulative capacity of device of type d at node v
f_{min}^d	Minimum operating frequency of device d
f_{max}^d	Maximum operating frequency of device d
Δf^d	Frequency step for DVFS on device d
F^d	Set of feasible frequency levels of device d
$P(f^d)$	Power consumption of a core of device d at frequency f^d
e_v	Power limit of node v
a	A microservice-based application
G^a	Directed acyclic graph representing application a
I_a	Total number of microservices of application a
r_i	Required processing cycles for the execution of service i
L_i	Communication delay limit of microservice i
$g_a \in V$	Data source node of application a
$\delta_{i,d}$	Coefficient to adjust the execution time of service i on device d
t_{i,d,f^d}	Execution time of service i on device d and frequency f^d
x_{i,v,d,f^d}	Equal to 1 if the i-th microservice is assigned to node v and a core of device d at f^d
y_{i,d,f^d}	Equal to 1 if the i-th microservice is assigned to a core of device d at f^d
M	Set of the application's execution paths
m_i	Equal to 1 if the i-th microservice is part of the execution path $m \in M$
q_m	Execution time of path $m \in M$
$\kappa \in M$	The application's critical path
t_i	Execution time of service i
T	The application's execution time
ε_i	Energy consumption of service i
E	The application's overall energy consumption

The energy consumption of service i, ε_i, can be calculated as the product of the power consumption with its execution time:

$$\varepsilon_i = \sum_{d=1}^{D} \sum_{f^d \in F^d} P(f^d) \cdot t_{i,d,f^d} \cdot y_{i,d,f^d} \quad (7).$$

Hence, the total energy consumption E for the application's execution is the sum of all individual services:

$$E = \sum_{i=1}^{I_a} \varepsilon_i \quad (8).$$

5.1.4.1 MILP Formulation

The objective function is the minimization of a weighted combination of the execution time and the total energy consumption for the application's execution. The weight coefficient w is used to control the relative importance of each objective:

$$\min w \cdot T + (1 - w) \cdot E \ (9).$$

Subject to the following constraints:

C.1 Each service $i = 1, \dots, I_a$ must be assigned to exactly one node and one device type configured at a specific frequency:

$$\sum_{v \in V} \sum_{d=1}^{D} \sum_{f^d \in F^d} x_{i,v,d,f^d} = 1, \forall i = 1, \dots, I_a \ (10)$$

C.2 Decision variables x and y are coupled by the following constraint:

$$y_{i,d,f^d} = \sum_{v \in V} x_{i,v,d,f^d}, \forall i = 1, \dots, I_a, \forall d = 1, \dots, D, \forall f^d \in F^d (11)$$

C.3 Node capacity constraint. Each service is assigned to one core on a processing device; therefore, the total services assigned to a device cannot exceed the available cores in the node:

$$\sum_{i=1}^{I_a} \sum_{f^d \in F^d} x_{i,v,d,f^d} \leq c_{v,d}, \forall v \in V, \forall d = 1, \dots, D \ (12)$$

C.4 Node power constraints. The cumulative active power of the processors should not exceed the power limit of the node:

$$\sum_{i=1}^{I_a} \sum_{d=1}^{D} \sum_{f^d \in F^d} P(f^d) \cdot x_{i,v,d,f^d} \leq e_v, \forall v \in V \ (13)$$

C.5 Services' communication limit. Every service must be assigned to a node that respects the delay limit with the data-generation node g_a:

$$l_{v,g_a} \cdot x_{i,v,d,f^d} \leq L_i, \forall i = 1, \dots, I_a, \forall v \in V, \forall d = 1, \dots, D, \forall f^d \in F^d (14)$$

Formulated as a Mixed Integer Linear Problem (MILP), the considered problem combines elements from the assignment problem and critical path analysis, inherently positioning it in the NP class.

5.1.5 Metaheuristic Mechanism

The formulated problem considers the deployment of one microservice-based application. Thus, a solution must be acquired for each incoming application in a timely manner, especially for time-critical workload. This motivates us to develop a meta-heuristic mechanism to tackle the increased problem's complexity.

5.1.5.1 Problem Decomposition

The primary problem is decomposed into two distinct yet closely linked sub-problems. The first problem involves mapping services to device types and frequency levels without accounting for the practical limitations of the existing infrastructure, such as node capacities and service delay requirements. This is referred to as the *Configuration Selection* problem. Its solution comprises the configuration tuple $[device, frequency] = [d, f^d]$ for each microservice, corresponding with the y_{i,d,f^d} variables.

The second sub-problem focuses on integrating this configuration within the actual infrastructure. Given the optimal y_{i,d,f^d} variables, the task is to determine the corresponding x_{i,v,d,f^d} variables. Thus, this second problem is termed the *Resource Allocation and Deployment* problem, as its solution obtains the final deployment tuple $[device, frequency, node] = [d, f^d, v]$ for each service.

By segregating the problem into a high-level assignment and a subsequent detailed mapping phase, the overall complexity is significantly reduced. Yet, this separation necessitates strategic considerations to ensure seamless interoperability between the two mechanisms. Instances may arise where solutions obtained from the Configuration problem are infeasible in the actual deployment due to inherent constraints. To mitigate such conflicts, it is essential to incorporate relevant high-level constraints within the Configuration problem. A complementary technique is to add complexity to the mechanism that addresses the Resource Allocation and Deployment problem so as to identify quality alternatives in case of misalignments with the exact solution of the Configuration problem.

5.1.5.2 Genetic Algorithm for the Configuration Selection Problem

Genetic algorithms are a great option when dealing with vast solution spaces that are not highly constrained [12]. For this reason, we choose to employ a genetic algorithm for the Configuration Selection problem. This way, the genetic can explore diverse solution spaces without the overhead of managing excessive constraints. Below we introduce the algorithm and the associated procedures.

In the context of genetic algorithms, chromosomes encode the solution to a problem, analogous to how real chromosomes encode the features of an individual. In our case, each chromosome represents a complete assignment of each microservice of an application to a device type and an operating frequency, that is the tuple $[[d_1, f^{d_1}], [d_2, f^{d_2}], \dots, [d_{I_a}, f^{d_{I_a}}]]$, where d_i is the selected device for microservice i and f^{d_i} represents the chosen operating frequency.

Given an encoded chromosome, the fitness function aims to evaluate the "goodness" of its genetic profile. Following the objective function (Equation (9)), we use the weighted combination of the execution time and energy consumption of the chromosome's configuration. Based on the assignment provided by the chromosome $[[d_1, f^{d_1}], [d_2, f^{d_2}], \dots, [d_{I_a}, f^{d_{I_a}}]]$, we can extract the execution time of each service based on Equation (4). The energy consumption can be straightforwardly computed using Equations $(5)$and (6). For the application's execution time, we need to identify the resulting critical path in order to apply Equations $(7)$and (8). The critical path is the longest path from any source node (with no incoming edges) to any sink node (with no outgoing edges), where the path length is the sum of the execution times of the nodes (services) along the path.

Algorithm 1: Critical Path Algorithm

Input: An application DAG $G(V, E)$, execution times $t[i]$ for each service $i \in V$
Output: critical path length $T_{critical}$

Initialize $L_i < -0 \; \forall i \in V$
$toporder < -TopologicalSort(G)$
foreach node i in $toporder$ ***do***
  If $in - degree(i) = 0$ *then*
    $L[i] < -t[i]$
  else
    $L[i] < -t[i] + max_{j \in pred(i)} L(j);$
  end
end
$T_{critical} < -\max_{i \in V} L[i];$
return $T_{critical}$

To this end, we employ a simple algorithm based on Dynamic Programming (*Algorithm 1*). The topological sort guarantees that every node is examined only after its predecessors, so that every node and every edge is visited exactly once in the procedure. For each node, its earliest completion time $(L[i])$ is the sum of its own completion time and the maximum of the completion times of its preceding nodes. After calculating for every node, the critical path is extracted as the maximum among these values. The algorithmic complexity of the algorithm is $O(V + E)$, where V is the number of nodes and E is the number of edges in the application's DAG.

Once the initial population is established, parent chromosomes must be selected to contribute their genetic material to the creation of offspring. We implemented a stochastic ranking-based selection method, which increases the likelihood of selecting high-fitness individuals while maintaining population diversity. Specifically, chromosomes are ranked in ascending order based on their fitness scores. The probability of selecting a chromosome at rank i is calculated as $\frac{X-i+1}{total}$, where X is the population size and $total = \frac{X \cdot (X+1)}{2}$. This approach ensures that higher-fitness chromosomes have a greater chance of being selected as parents without entirely eliminating less fit individuals, thereby preserving genetic diversity within the population.

For the crossover operation, we employed uniform crossover to ensure a thorough mix of the selected parent's characteristics in the offspring. Every mating operation results in two offspring: The first inherits each gene (device and frequency configuration for a service) with $a\%$ probability from the first parent and with $(1-a)\%$ from the second parent, while the exact opposite applies to the second offspring. This aims to maintain the "goodness" of each parent, while still providing mix-up for enhancements.

Additionally, each offspring undertakes a mutation either in device, frequency, or both compartments of each gene. First, an initial mutation probability for device β_d and frequency β_f is set. However, in order to exploit the evolved generations, this probability decays at a rate of$1-\frac{c}{N}$,where c is the current generation and N is the total number of generations.

Algorithm 2: Genetic Algorithm

Input: Initial chromosome population, population size X, number of generations N, elites E
Output: Best resulting chromosome

foreach generation $n = 1{:}N$ **do**
  Rank chromosomes based on their fitness;
  Extract top E elites and copy them to the next gen;
  Initialize next generation with E elites
  while $current\ population\ size < X$ **do**
    Perform rank-based selection;
    Perform crossover on selected chromosomes and produce offspring
    Add offspring to the next generation pop;
  end
  Perform mutation operation on the offspring in the next generation;
end
return top-performing chromosome of generation N;

Finally, we utilized the elitism feature to "save" good solutions throughout generations, while making sure that the best-fitting chromosome at one generation is no worse than that of the previous one. This means that the highest-performing chromosomes are copied to the next generations unchanged. The pseudocode for the algorithm is presented in *Algorithm* 2.

5.1.5.3 Best-fit heuristic algorithm for the Resource Allocation and Deployment problem

Upon receiving the best configuration for the application's services (y variables), the heuristic attempts to map this configuration into the infrastructure. The algorithm performs the assignment for each service sequentially. After selecting a service i and its configuration $[d_i, f^{d_i}]$, it first identifies nodes with the required capacity of the selected device. Then, nodes that do not respect the delay-limit of the service are pruned. The algorithm proceeds to place the service at the node which has the largest "power-capacity", i.e., the one which is percentage-wise the furthest from meeting its power constraint. This ensures a fair load-balancing across the infrastructure and avoids over-stressing specific nodes. The algorithm then proceeds with the following microservice, until all are placed.

However, instances arise wherein the heuristic cannot find a feasible mapping for the specified device and frequency level. In this case, it searches for an alternative combination of device and frequency that produces the "closest" objective value compared to the original. This way, the final deployment of the application might not be identical to the one suggested by the genetic algorithm, but it will effectively follow its effectiveness in terms of objective value.

Figure 4 shows the interplay between the developed genetic and heuristic mechanisms. Initially, the heuristic algorithm, utilizing telemetry agents distributed across the infrastructure, provides the genetic algorithm with real-time information on the infrastructure's status. This includes data on delays relative to the data source node, resource availability, power constraints, and other relevant parameters.

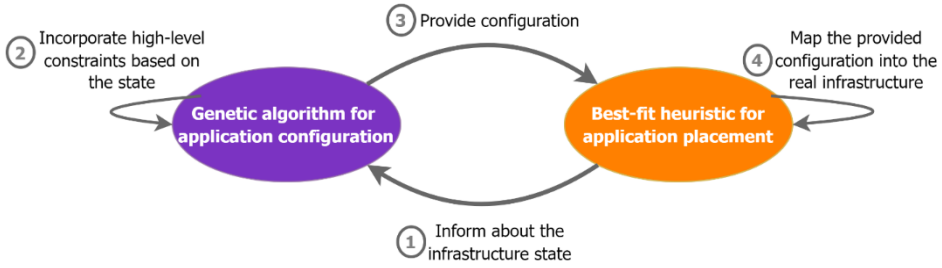


Figure 4: Interplay between the developed mechanisms.

In response, the genetic algorithm integrates these constraints by modifying the initial population and adjusting mutation rates to promote the generation of feasible solutions. For example, considering a service's delay limit and the availability of devices within nodes, the genetic algorithm initializes solutions that incorporate only the available devices for that service. Moreover, if the orchestrator reports (utilizing the distributed EMPYREAN telemetry agents) that a resource is depleted (either malfunctioning or fully occupied), the genetic is informed to exclude that resource in its solutions. Once it identifies the best chromosome, this configuration is relayed back to the heuristic algorithm. The EMPYREAN orchestrator then performs the final resource binding, effectively deploying the application within the Association based on the optimized configuration (by communicating its decisions with the local orchestrators).

5.1.6 Evaluation

5.1.6.1 Experimental Setup

We conducted a series of experiments to showcase the efficacy of the developed mechanisms along with the trade-offs in the objectives for different weighting coefficients w. The genetic algorithm was implemented in Python, utilizing the *NetworkX*[1] library for the realization of the

[1] https://networkx.org

application's DAG. The heuristic algorithm along with the infrastructure simulation were developed in MATLAB. The experiments were conducted on a Ryzen-7 32 GB RAM PC.

Table 2: Base and Max Frequency of resources.

Processor Model	Base Frequency (Ghz)	Max Turbo Freq. (Ghz)
AMD EPYC™ 9965	3.5	5
AMD Ryzen™ AI 9	2	5.1
AMD Ryzen™ 5 PRO 8500G	3.5	5
Intel Xeon W-11955M	2.6	5
ARM Cortex-A76	--	3.3

We considered 10 device types ranging from edge microprocessors (e.g., NVIDIA Jetson Series) and edge computer devices (e.g., Intel Core™ i7 series) to high-performance server processors (e.g., Intel Xeon, AMD EPYC). For each device, we set the f_{min}^{d} to 50% of the disclosed *base frequency* (underclocking), while we allow frequency tuning up to the *max turbo frequency* specified by the manufacturer. Moreover, most processors allow for a 100 MHz frequency step between the lowest and highest frequencies. However, we adopt a coarser approach by creating 10 frequency levels for each device d, normalizing the step Δf^{d} accordingly. Some of the real-world processing devices [15][21] from which we drew the experimental values for the base and max frequencies are presented in Table 2. Regarding power consumption, we identified measurements on Intel processors [20] and Nvidia edge devices [15] (Table 3), and used them as a guide to estimate power consumption for the rest of the considered devices, combined with the formula in [11]. Generally, activating a server core requires more power than a core of a microprocessor in the edge.

Table 3: Frequency and associated power consumption of resources.

Processor Model	Base Frequency (GHz)	Base Power (W)	Max Turbo (GHz)	Max Power (W)
Core i9-11900K	3.5	125	5.2	251
Core i7-10700	2.9	65	4.9	224
Core i3-10000T	3	35	3.8	55
Jetson Nano 0.9	0.9	5	1.5	10
Jetson AGX Xavier	1.2	10	2.27	30

The infrastructure was modeled as a 3-layered topology. The near-edge layer, the most proximal to end-users, comprises a total of 30 nodes, each possessing between 1-5 micro-processing devices. The delay between near-edge nodes and end-users was normalized in the [0.5,2] delay units (d.u.) range. The far-edge layer, positioned amidst the urban areas and the remote Data Centers, includes 10 nodes, each of which hosts 5-10 devices of medium capabilities. The delay for this layer is set to [3,5] d.u. Finally, the cloud layer consists of 2 nodes representing the core Data Centers, equipped with 100 high-end server processors. The power limit was set to 50% of the maximum achievable power (where all cores work at max frequency) for near-edge nodes, while for the far-edge and cloud the limit is 70% and 80% accordingly.

Microservices were assumed to demand between 0.1 and 10 Giga-Cycles for their processing, reflecting their heterogeneity based on their scope. Coefficient $\delta_{i,d}$ was set in the [1,2] range, with some services exhibiting minimal discrepancies between devices, while others benefit more from the advanced architecture of the server processors compared to edge.

In all experiments, the values of application delay and energy consumption were normalized in the [0,1] interval utilizing the max-min method. This approach makes the weight parameter more intuitive; for example, setting w=0.5 implies that both metrics contribute equally to the objective.

5.1.6.2 Evaluation Results

First, we evaluated the performance of the genetic algorithm. To this end, we generated an application comprising 100 microservices. The corresponding DAG produced by the *NetworkX* is illustrated in Figure 5. The population size of the genetic algorithm was initialized to 100 chromosomes. Regarding mutation, we set the initial probability for both device and frequency mutations for each gene to 15%. Moreover, elitism was employed, retaining the top 5% of the population in each generation.

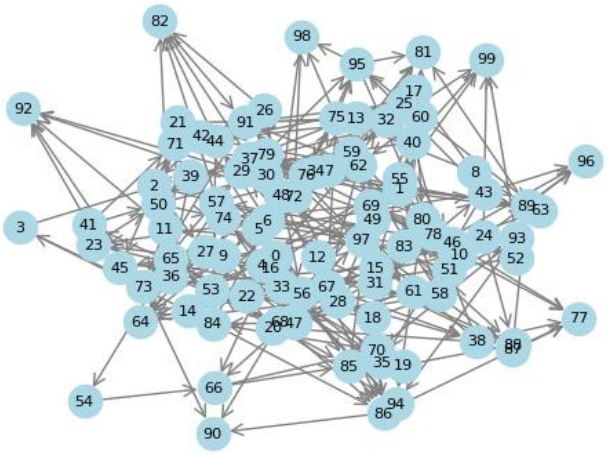


Figure 5: Example of generated DAG.

Figure 6 presents the genetic algorithm's convergence simulation results across different weight coefficients w. After several runs, we chose to terminate the genetic algorithm at 2000 generations, as it produced the best balance between performance and execution time. The resulting optimality gaps were 2.8%, 3.9% and 2.3% for $w = 0.1$, $w = 0.5$ and $w = 0.9$, respectively. For $w = 0.5$, execution time and total energy consumption are considered equally, which complicates the problem. The algorithm exhibited monotonic convergence, facilitated by elitism, ensuring that the best-fit individual in each generation was at least as effective as in the preceding generation. Notably, we excluded single-objective optimization (i.e., $w = 0,1$), as it is not of practical relevance to completely neglect either execution time or energy consumption in a real-world deployment and renders the solution trivial.

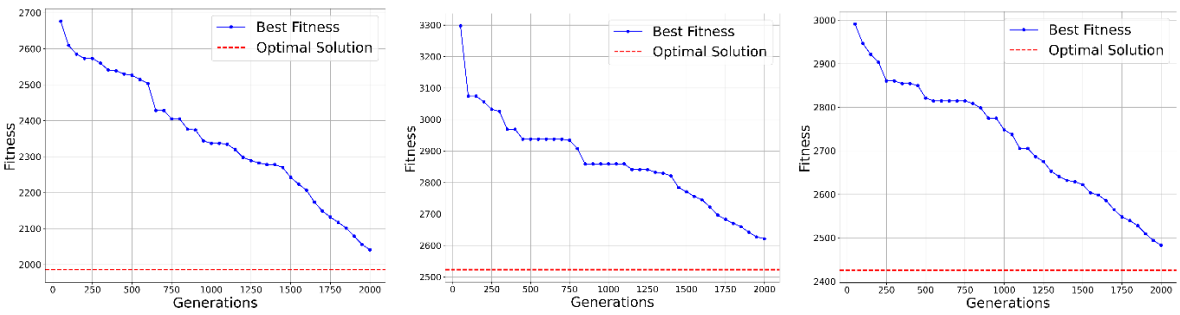


Figure 6: Convergence of the genetic Algorithm for w=0.1 (left), w=0.5 (middle) and w=0.9 (right).

Regarding the execution time, by parallelizing the chromosome evaluation, the genetic algorithm clocked in at 4.21 seconds on average, while the optimal solver based on the PULP library averaged at 513 seconds. An initial deployment configuration lasting a few seconds for a 100-microservice application is deemed acceptable, as it remains comparable to other required steps, such as fetching container images, building them, and starting containers.

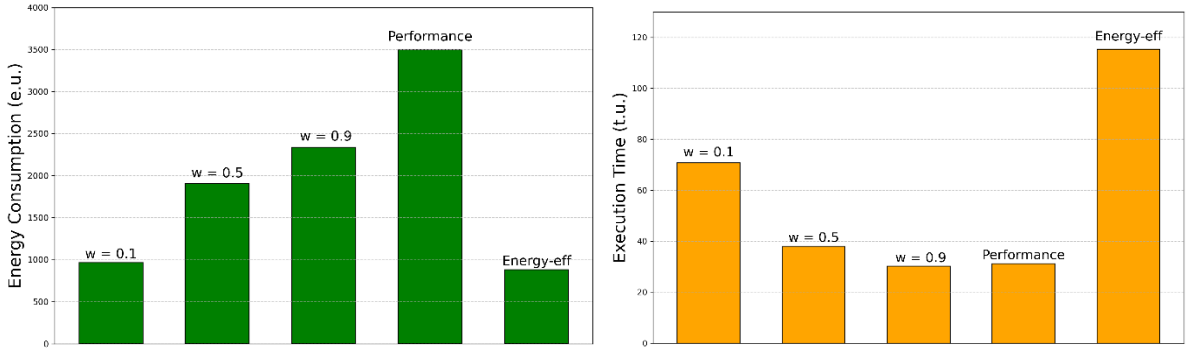


Figure 7: Comparison of Energy consumption (left) and Execution time (right) across mechanisms.

Next, we employed two baseline algorithms to contrast their results with our proposed mechanism: i) The "Performance" policy aims to minimize the application's execution time by greedily assigning each microservice to the ($[device, frequency]$) pair that offers the lowest execution time within the infrastructure and ii) The "Energy efficiency" policy focuses on minimizing energy consumption, by greedily assigning each service to the $[device, frequency]$ pair that offers the lowest energy consumption.

Figure 7 presents the execution time, measured in time units (t.u.) along with the total energy consumption, measured in energy units (e.u.), across the different mechanisms. We tested three different weight coefficients for our mechanism, $w = 0.1, w = 0.5\ and\ w = 0.9$, as indicated above the corresponding bars. The lowest energy consumption was achieved by the Energy-efficiency policy, outperforming our mechanism when utilizing weight $w = 0.1$ by \$8.2%\$. However, in terms of execution time, the Energy-efficiency policy resulted in a 51.1% increment.

Interestingly, our mechanism, when tuned with $w = 0.9$, outperformed the Performance policy in terms of execution time by 3.2%. This is because our developed metaheuristic calculates the resulting critical path and thus optimizes the execution time of the application as a whole. In contrast, the Performance policy potentially wastes resources due to its greedy nature. Additionally, the configuration provided by our mechanism managed to cut-down energy consumption by 31.5% compared to the Performance Policy. By fine-tuning the weight coefficient, our mechanism can intelligently balance objectives, achieving enhanced performance without excessive energy consumption ($w = 0.9$), improved energy efficiency without significant performance degradation ($w = 0.1$), or a balanced approach ($w = 0.5$).

Finally, Figure 8 presents the distribution of services across the infrastructure layers for each examined mechanism. The Energy Efficiency policy opted for the near-edge layer, exploiting the inherent lower-power and lower-frequency devices to minimize energy consumption. Our mechanism, configured with $w = 0.1$, also predominately utilized the near-edge, but still deployed some critical services on the upper tiers to enhance performance. The Performance policy favors the high-end systems of the far-edge and the cloud-layer to reduce execution time. Our mechanism, when tuned with $w = 0.9$, utilized more of the cloud layer compared to Performance. However, it also employed part of the near-edge to enhance energy efficiency on non-critical parts of the application.

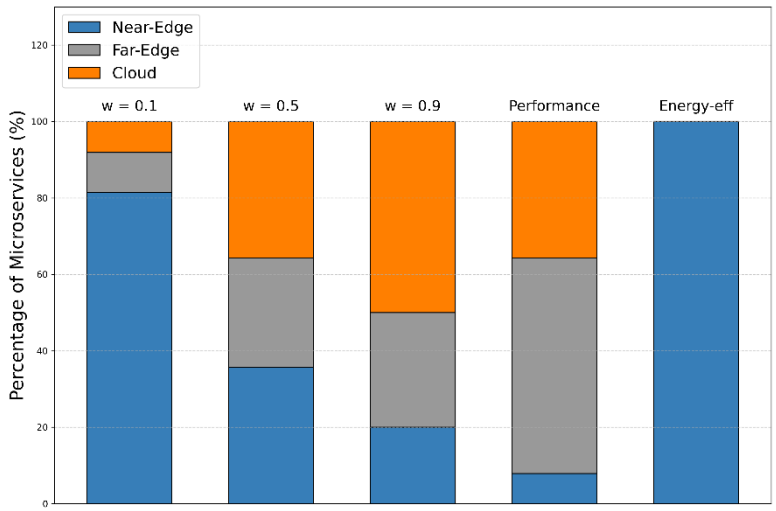


Figure 8: Service distribution across infrastructure layers.

5.1.7 Conclusion

This work introduced a DVFS-enabled, critical-path-aware mechanism for deploying microservice-based applications over the edge-cloud continuum. We modeled the problem as a MILP, targeting to optimize a weighted combination of the application's execution time and the total energy consumption. A novel two-phase heuristic approach was developed to tackle the problem's inherent complexity, comprising a genetic algorithm for the configuration problem, followed by a best-fit heuristic for the resource allocation and placement problem.

Our experiments highlighted the efficiency of our proposed method: By properly fine-tuning the weight coefficient, our mechanism can intelligently configure and deploy applications, leveraging the heterogeneous devices across the infrastructure to meet performance and energy objectives. As a future direction, we plan to incorporate the delays between infrastructure nodes into the problem formulation and develop real-world testbeds for evaluation.

5.2 Risk-Aware Resource Allocation in Edge Computing Using Stochastic Forecasting

5.2.1 Abstract

Edge computing brings processing closer to data sources, reducing latency and bandwidth usage for modern applications. However, the limited capacity of edge resources and volatile nature of workload demands create significant challenges for efficient resource management, often leading to resource underutilization. In this work, we propose a speculative resource allocation framework supported by stochastic workload forecasting, inspired by the Black-Scholes financial model. This framework dynamically assesses the risk associated with fluctuating demands over different time windows and proactively aligns resource allocation based on the performed risk assessments. The outcomes of this model drive a multi-objective resource allocation mechanism that dynamically manages resources, speculatively aligning differing workload demands when placing them within a node, optimizing key performance metrics such as latency, infrastructure utilization, cost-effectiveness, and potential application disruptions during execution. Our approach does not require training, making it more adaptable to fluctuating demands compared to machine learning-based methods. Through simulations we demonstrate that our framework improves performance and resource utilization, providing a scalable, responsive, and cost-effective solution that benefits both end-users and operators.

5.2.2 Introduction

Edge computing has emerged as a critical paradigm to meet the increasing demands for low-latency and bandwidth-efficient processing, especially in modern applications driven by IoT, AR/VR, autonomous systems, and real-time analytics. It brings processing and storage capabilities closer to the data source and helps overcome the limitations of cumbersome centralized cloud computing [24]. However, despite its advantages, edge environments face significant constraints, primarily due to limited resource capacity. Coupled with the volatile and unpredictable nature of workload demands, efficient resource management becomes a challenging problem. Existing resource orchestration mechanisms often struggle to optimize resource usage, resulting in poor system performance and increased operational costs [25].

A key issue in edge resource management arises from the overprovisioning and underutilization of resources, primarily caused by reactive resource allocation schemes that fail to anticipate dynamic workload patterns. Traditional methods often allocate resources based on worst-case assumptions [26] or rely on user-provided estimates [27], approaches that are either too conservative or unresponsive to real-time changes in application demands. Consequently, this results in wastage of resources; studies have shown that in many data centers, up to 76% of CPU resources and 26% of memory resources remain idle for extended periods [28],[29].

Additionally, proactive solutions, many of whom rely on machine learning (ML) and data-driven models, face their own limitations. These approaches require extensive training on historical data, making them rigid and slow to adapt to dynamic and previously unseen workloads without time-consuming retraining processes that undermine their real-time applicability [30]. This creates a fundamental trade-off between adaptability and efficiency, where resource orchestrators must choose between fast, reactive models with limited foresight or proactive models with significant retraining overheads.

Hence, there is a gap in edge resource management: the need for an adaptable, risk-aware resource allocation mechanism that can anticipate demand volatility without relying on static assumptions or time-consuming retraining. This mechanism must dynamically allocate resources based on real-time workload behavior, while minimizing the risks of both under- and over-provisioning, and doing so without significant computational overhead [31]. In this work, we draw inspiration from the financial sector, specifically, the Black-Scholes model [32], widely used in options pricing and risk management. The analogy between financial markets and edge resource management is compelling: in both cases, there is a need to account for uncertainty and volatility in dynamic environments. As the Black-Scholes model helps traders assess the risk of price fluctuations, a similar approach can assess the risk of workload fluctuations.

Our mechanism emphasizes risk quantification and mitigation, employing a stochastic framework to estimate the likelihood that resource demand will exceed available capacity within a given time window. This enables speculative resource allocation while managing the inherent trade-offs between the risk of resource saturation and the cost associated with the underutilization of resources. The proactive risk management approach enables (i) better foresight than reactive models and (ii) avoids the retraining process of ML, making it a suitable choice in the volatile and resource constrained edge computing environments.

5.2.3 Related Work

Research efforts in the field of edge-cloud resource orchestration have mainly focused on forecasting workload demands and optimizing resource allocation, providing solutions to address the complexities of dynamic, interconnected, and distributed computing environments.

Various methods have been proposed for forecasting workload demands. Statistical models such as ARIMA (Auto Regressive Integrated Moving Average) have been the traditional approach [33]. These models are widely used for time-series processing and perform well under stable data conditions and consistent patterns. However, their main drawback is their inability to handle non-linear and volatile workloads commonly found in edge computing environments resulting in inaccurate predictions and suboptimal allocation. For this reason, ML methods have been employed to enhance the accuracy of workload forecasting and provide the needed adaptability through data driven approaches either alone or combined with statistical models. The authors in [34] proposed a hybrid model that combines statistical forecasting with neural networks in one framework that utilizes a statistical model to generate initial predictions, which are then refined by a neural network for enhanced accuracy.

Time series workloads are particularly well-suited for Recurrent Neural Networks (RNNs) and Long Short-Term Memory Networks (LSTMs) [35]. Recently, Transformers have taken over many tasks in which RNNs were used traditionally with the authors in [36] proposing an Adversarial Transformer model for cloud workload forecasting, which outperformed previous LSTM methods in inference time and prediction accuracy. To address the variability of applications and the microservices comprising them across different devices, the authors in [37] proposed a Transformer model that groups them based on various criteria. Nevertheless, this approach necessitates multiple resource intensive models, and similar to all ML models, they require frequent fine-tuning with updated data to maintain their high performance [30].

Another distinction in forecasting approaches lies between the previously mentioned proactive methods and reactive ones. The latter, such as Kubernetes' horizontal pod autoscaler, adapt to changes in workload demand by adjusting the number of instances based on current traffic or CPU usage [38]. Efficient modifications have been proposed [39] but while they provide a real-time response to shifts in demand, they are inherently reactive, meaning they scale resources after performance has degraded also incurring extra delays.

The second focus of edge cloud resource orchestration, i.e., resource allocation, is a field in which there have been many optimization efforts, with the most advanced of these also leveraging forecasting insights to facilitate informed decision-making [40]. Some techniques emphasize execution speed and simplicity employing greedy heuristic methods in its offline version [41]. In online scenarios where user requests arrive constantly, the continuous horizon nature of the task scheduling problem makes it an ideal case for Reinforcement Learning [42], a subcategory of ML, where reactive agents dynamically interact with the infrastructure to maximize service profitability and Quality-of-Service (QoS), that is however subject to the same drawbacks as ML. In [43] the authors propose a multi-layer hierarchical system for joint online task scheduling, resource allocation and caching. A Double Deep Q-Network (DDQN) approach, proposed in [44], dynamically adjusts the resource allocation decisions to changing resource statuses and workloads, ensuring optimal decision-making in real-time. Multi-Agent methods have also been introduced to decentralize orchestration, federating allocation decisions and creating a flexible framework that efficiently utilizes the distributed infrastructure [42].

The works described underline the complexity of managing distributed infrastructures, with dynamic resource allocations mechanisms adaptively responding to the varying application requirements. While effective, ML mechanisms often rely on data-driven forecasting methods to guide their operations. Their effectiveness and applicability in real-world cases is thus constrained by the ever-changing data patterns requiring periodic recalibration with scarce data in a time consuming and resource intensive process. Moreover, the volatile needs of such environments demand intelligent solutions that efficiently handle dynamicity in real-time scenarios but operate under strict time constraints in distributed settings without compromising performance, thus necessitating a mix of both reactive and proactive policies. Hence, our approach is explicitly designed to handle dynamicity by employing stochastic forecasting inspired by financial models, enabling accurate allocation with little overhead, using current usage data, and also reacting to real-time events by rearranging resources in an efficient manner.

5.2.4 System Model

We model the edge-cloud infrastructure as a distributed computing environment that spans across multiple nodes, ranging from near edge devices to centralized cloud resources (Figure 9). This infrastructure is represented as an undirected weighted graph $G = (V, E)$, where the set of vertices V represents the locations where computing resources are placed, and the edges $e \in E$ represent the network connections between these locations. The weight of each edge d_e corresponds to the network latency delay between these nodes, a parameter that needs to be considered during the allocation of the workload demands to resources. Each node $v \in V$ represents a physical computing node, ranging from personal devices (Raspberries, Home Servers etc.) to small- and large-scale data centers. These nodes are characterized by a tuple $\tau_v = [c_v, o_v, b_v]$; (i) the CPU capacity c_u of the node, measured in the number of cores available for processing, (ii) the operating cost per CPU core used o_u, and (iii) the bandwidth cost of v b_v per bandwidth unit (b.u.) transferred.

The workload is modelled as a set of application demands A, each consisting of multiple microservices. Applications $a \in A$, are represented as directed acyclic graphs $G_a = (V^a, E^a)$, where each node $v^a \in V^a$ corresponds to a microservice and the edges E^a represent the dependencies between those microservices. The weight of an edge $w^a_{i,j}$ represents the maximum acceptable communication delay among microservices i and j of application α. Each microservice requires m_{v^a} b.u. to be transferred to a node prior to execution, prone to bandwidth utilization billings.

Time in our system model is divided into discrete fixed duration periods. Applications arrive dynamically, with each application $a \in A$ having a start time and a duration, defining the number of periods it will run for. Each microservice has specific computational requirements for each distinct period expressed as $S_{a,i,t}$, which represents the CPU demand for the microservice during time period t . This demand fluctuates over time, and we represent it as a time series over the different periods to capture its dynamic nature.

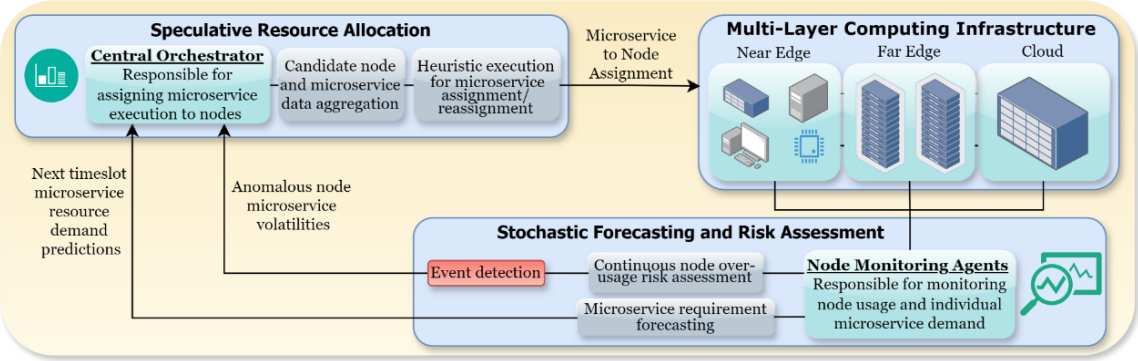


Figure 9: The allocation of resources across the distributed edge-cloud infrastructures.

5.2.5 Speculative Application Resource Allocation through Stochastic Risk Assessment

In this section, we introduce the speculative resource allocation mechanism enabled by stochastic risk assessment, presented in Figure 9. Designed to manage the uncertainty of workload demands in edge-cloud environments, it addresses these challenges by proactively adjusting resources based on workloads' risk to surpass a pre-defined threshold. This can be set by system administrators or service-level agreements to ensure QoS and avoid resource saturation. The mechanism directly utilizes the workload's volatility, allowing dynamic resource reallocation before demand spikes to nodes that minimize the risk of over-provisioning. When placing the workload demands to nodes the assignment is done in a manner that minimizes the cumulative risk of resource exhaust within the node. This combination ensures efficient, real-time allocation, while accounting for risk, increasing utilization and minimizing service disruptions.

The process operates in two alternating stages. The first stage involves the forecasting mechanism, which calculates both the expected growth rate $\mu_{\alpha,i}$ and the volatility parameter $\sigma_{\alpha,i}$ for each microservice i in application a over a past window T. While volatility $\sigma_{\alpha,i,t}$ quantifies the uncertainty in the microservice's CPU demand $\varepsilon_{a,i,t}$ over time, $\mu_{\alpha,i,t}$ captures the anticipated trend in CPU usage. Using these parameters, the system estimates the future CPU requirements within the next time window, incorporating a given risk threshold R to balance resource allocation with potential demand fluctuations. This forecasting step repeats every T periods to ensure the system continuously adapts to changing workloads. By modifying the forecasting horizon T and changing the frequency of execution for the resource allocation algorithm we can define the granularity of our predictions, managing a trade-off between forecasting accuracy and computational intensity. Performing the risk assessment and workload forecasting locally, the system minimizes real-time telemetry data transfers, only sending critical estimates and risk levels to the orchestrator enhancing reaction speed and reducing communication overhead. The combination of joint risk assessment and volatility-based workload assignment ensures a steady temporal resource allocation, increasing infrastructure utilization and reducing potential QoS degradations.

In the second stage, between forecasting windows, the Speculative Resource Allocation (SRA) mechanism monitors node utilization and assesses the risk of overuse based on the current volatility $\sigma_{\alpha,i,t}$. If the risk of exceeding the available resources surpasses the defined QoS threshold R, the system notifies the central orchestrator to reallocate resources preemptively. This enables proactive, multi-objective optimization that accounts for overprovisioning risks as well as other criteria like end-to-end latency and cost efficiency.

5.2.5.1 Stochastic forecast and risk assessment mechanism

We introduce the stochastic forecasting and risk assessment mechanism designed to monitor individual microservices and nodes with minimal computational overhead. This mechanism provides a QoS, represented by the probability that actual CPU demand will not exceed the allocated resources. To achieve this, we model the CPU demand of microservices $S_{a,i,t}$ using Geometric Brownian Motion (GBM), a widely used method for representing fluctuating variables. The GBM model for the CPU demand is defined by the equation:

$$dS_{a,i,t} = \mu_{a,i,t} S_{a,i,t} dt + \sigma_{a,i,t} S_{a,i,t} dW_{a,i,t} \quad (1)$$

where $\mu_{a,i,t}$ the drift rate (expected growth rate of CPU demand), $\sigma_{a,i,t}$ the volatility (standard deviation of CPU demand), $W_{a,i,t}$ a Wiener process for a microservice i of an application a as calculated at time t.

This modelling choice is justified by the similarities between distributed computational systems and financial markets: both operate continuously, experience uncertainties from external factors such as hardware malfunctions and fluctuating user demand [8] and generate data that can be leveraged by data-driven approaches for resource orchestration. Just as stock prices fluctuate based on trading activity, CPU demand for applications and their microservices is driven by user behaviour. Thus, we can predict CPU demand over a prediction horizon H and allocate resources, accordingly, minimizing the risk of under-provisioning. In this analogy, the CPU demand of a microservice reflects the volatility $\sigma_{a,i}$, while the allocated resources act as a threshold $K_{a,i,t+H}$ ensuring adequate provisioning. Following properties of GBM, the logarithmic ratio of future CPU demands $K_{a,i,t+H}$ to current demands $K_{a,i,t}$ follows a normal distribution.

$$\ln\left(\frac{K_{a,i,t+H}}{K_{a,i,t}}\right) \sim N\left(\left(\mu_{a,i,t} - \frac{\sigma_{a,i,t}^2}{2}\right)H, \sigma_{a,i,t}^2 H\right) \quad (2)$$

Therefore, the required resources $K_{a,i,t}$ that need to be allocated at time t and a prediction horizon H to meet the QoS threshold R as given in [45] is presented below:

$$K_{a,i,t} = S_{a,i,t} \cdot e^{-N^{-1}(R)\cdot\sigma_{a,i,t}\sqrt{H} + \left(\mu_{a,i,t} - \frac{\sigma_{a,i,t}^2}{2}\right)\cdot H} \quad (3)$$

where $N^{-1}(R))$ is the inverse cumulative distribution function of the standard normal distribution at probability R. Eq. (3) allows us to determine the resource level that must be allocated for microservice i of application a to ensure that, with probability R, the CPU demand will not exceed the allocation over the prediction horizon H. In an analogy to the

Black-Scholes model, the QoS threshold R mirrors the probability that an option will expire in-the-money, and the model's prediction horizon H is akin to the time-to-expiry.

To assess the risk $R_{u,t}$ at time t that the CPU resources of a node u may be insufficient until time $t + H$, it is essential to define the aggregated expected growth rate $\mu_{u,t}$ and aggregated volatility $\sigma_{u,t}$ of CPU demand on node u at time t . Let node u host a set of applications $M_{u,t}$ a and a set of microservices $M_{u,t}$. Each microservice $M_{u,t}$ in application a assigned to node u has an expected CPU demand growth rate $\mu_{a,i,t}$ and volatility $\sigma_{a,i,t}$. Assuming that the CPU demands of microservices across different applications are independent, while microservices within the same application may exhibit dependencies, the aggregated parameters for node u are derived as follows:

$$\mu_{u,t} = \sum_{a \in A_{u,t}} \sum_{i \in M_{u,t}(a)} \mu_{a,i,t} \tag{4}$$

$$\sigma_{u,t}^2 = \sum_{A_{u,t}} \sum_{i \in M_{u,t}(a)} \sigma_{a,i,t}^2 + 2 \cdot \sum_{A_{u,t}} \sum_{i,j \in M_{u,t}(a)} Cov\,(S_{a,i,t}, S_{a,j,t}) \tag{5}$$

Note that the volatility calculated by Eq. (5) considers the covariance for each pair of dependent microservices, i.e., the microservices of the same application to account for their joint variability. Using the aforementioned parameters, the risk $R_{u,t}$ that node's u capacity will be exceeded is quantified as:

$$R_{u,t} = 1 - N\left(\frac{\ln\left(\frac{K_u}{S_{u,t}}\right) - \left(\mu_{u,t} - \frac{{\sigma_{u,t}}^2}{2}\right) \cdot H}{\sigma_{u,t}\sqrt{H}}\right) \tag{6}$$

5.2.5.2 Speculative Cloud native application Resource Allocation mechanism

The proposed SRA mechanism leverages the output of the forecasting mechanism for task assignment across the distributed infrastructure. At the start of each period, the SRA mechanism executes a sequence of operations: (i) Resource Deallocation: The expired resources allocated to applications are released, updating the respective nodes' resource availability; (ii) Resource Allocation: a greedy best-fit heuristic is employed, to allocate resources for new applications requests using the predictions provided. The greedy best-fit heuristic processes application demands sequentially with two modes of operation. In the case of node overutilization, the mechanism evaluates the individual resource requirements only for microservices affected by the event, while it assesses all the applications' microservices when updating the total allocation using new predictions. For each microservice, it identifies candidate nodes that possess the necessary CPU capacity and computes an evaluation metric that encapsulates the node's suitability. The value of this metric is the weighted sum of (i) the expected communication latency from hosting the microservice; (ii) the cost of utilizing the node; and (iii) the risk associated with potential resource saturation. The nodes are then sorted in descending order based on their heuristic scores.

Depending on the mode of operation the mechanism then attempts to assign the affected microservices or all microservices to a node in the ranked list, moving on to the next if a node's resources fall short or the application's latency constraints are not satisfied. The allocation process continues until it has either successfully assigned all microservices or until the exhaustion of the candidates list, in which case there is no viable assignment scheme with the current allocation and a backtracking procedure is initiated, systematically freeing up resources for all microservices related to the applications in question. Resource allocation is then reattempted with a greater number of candidate nodes, allowing for greater flexibility while giving priority to the node a microservice is initially hosted at to minimize the number of re-locations of microservices. In this way, the allocation decision takes the forecasted resource utilization patterns into consideration for the already served applications. A high-level overview of the SRA mechanism is provided in the pseudocode in Figure 10.

Speculative Resource Allocation Mechanism
Input: Nodes, Applications, Risk Threshold
Output: AllocationStatus, NodeStates

```
for each timeslot t do:
  for each node v in V do:
    for each application a in allocated with end time ≤ t do:
      Free allocated resources
      Remove application a from allocated
    end for
  end for
  for each application a starting at time slot t do:
    for each microservice i in a do:
      candidates = [for v in V with enough resources]
      Calculate heuristic score
      Order nodes based on objective function
      if candidate_node do:
        Allocate resources of candidate node
        Append microservice to served
      else (no candidate node):
        for the previous allocated microservice do:
          Free Resources (prev_micro)
          Try to allocate resources to freed microservices
      end if
    end for
  end for
  for each node v in V where R_v,t > Risk Threshold do:
    for microservice i in allocated in v do:
      Reallocate Microservice i
    end for
  end for
end for
```

Figure 10: The pseudocode of the speculative resource allocation mechanism.

5.2.6 Simulation Experiments

5.2.6.1 Simulation Setup Description

For our simulation experiments, we considered a fully connected network topology consisting of 47 nodes to comprehensively evaluate the performance of the SRA mechanism across a distributed edge-cloud infrastructure. This setup allowed us to mimic a realistic and scalable edge cloud continuum infrastructure. The specific characteristics of each node type, including the number of nodes, CPU units, latency metrics in latency units (l.u.), operating and bandwidth costs in cost units (c.u.), are detailed in Table 4. When a reallocation is performed, we assume an increased cost of execution equal to 10 times the operating node's cost to penalize urgent service disruptions.

Table 4: Infrastructure characteristics.

Node Type	# of nodes	# of CPU units	Latency Units (l.u.)	Operating Cost (c.u.)	B/W cost (c.u)
Near Edge	40	[4,8]	[3-10]	[4-6]	[5-10]
Far Edge	6	[80-120]	[20-50]	[2-3]	[2-5]
Cloud	1	500	100	1	2

For the user workloads, we employed the Alibaba Cluster Trace [46], which captures data from a production cluster over a 12-hour period, to test our framework's predictions' applicability on realistic data. Average resource demands in 5-minute intervals were extracted to accurately model fluctuations. The microservices specifications were randomly generated based on a uniform distribution to create a versatile framework for various applications. Each application comprised 1 to 6 microservices, with their computational intensities—low, medium, or high—assigned probabilities of 0.45, 0.30, and 0.25, respectively. This setup captures the heterogeneity of possible workloads while simultaneously utilizing real demand from the Alibaba trace. Dependencies among microservices were established randomly with a probability of 0.5 to simulate necessary interactions within the applications. Latency communication constraints ranged from 5 to 50 l.u. among interdependent microservices. The proposed mechanisms were developed in Python and simulations were conducted on a Ryzen 7-32GB RAM PC.

5.2.6.2 Performance Analysis

Initially, we examined the SRA mechanism's efficacy. We gradually increased the model's risk limits R from 1% to 5% and compared their outcomes with those of a baseline model that operates based on a static worst-case resource allocation. The metrics of focus were the Utilization Ratio, reflecting the ratio of actively utilized to total allocated resources, and the number of relocations, indicating the frequency of microservice migrations due to resource unavailability. Our findings, depicted in Figure 11, reveal that the SRA mechanism significantly enhances resource utilization under all forecasting scenarios improving utilization upon the

baseline model by up to 1.59 times, underscoring the effectiveness of our forecasting mechanism. Moreover, it achieves this efficiency while concurrently reducing the number of relocations by up to 78%. Notably, as the risk limit increases, the following trade-off emerges: a higher risk limit leads to greater resource allocation upfront, and less efficient utilization. On the other hand, it diminishes the need for subsequent relocations. This adaptability is crucial when applications with varying requirements and constraints are served, allowing for a flexible framework that can achieve high QoS while maintaining efficiency.

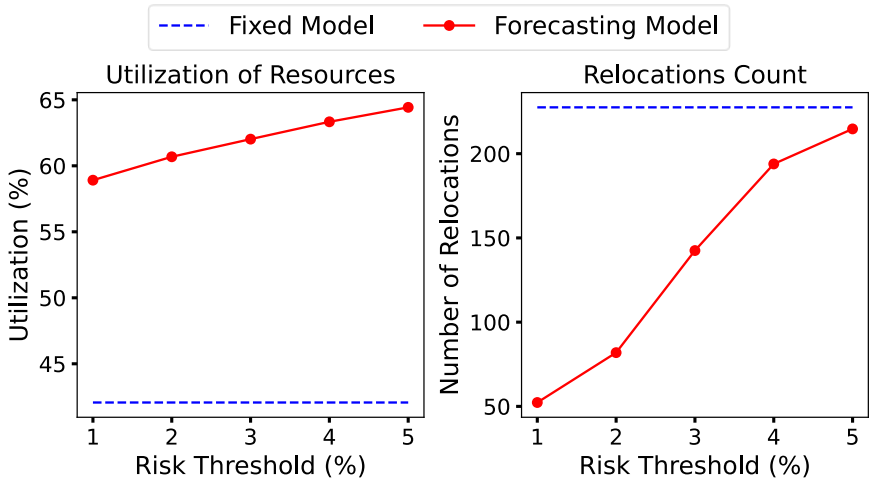


Figure 11: The effect of shifting forecasting risk limit R in the utilization of allocated resources and the number of reallocations of services.

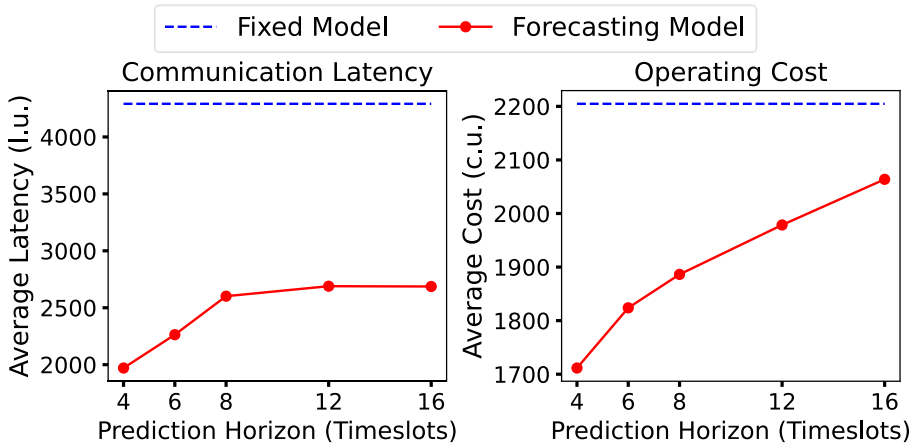


Figure 12: The effect of shifting the prediction time horizon H in the operational cost and the end-to-end application latency.

Next, we examined the impact of the forecasting horizon H on the total cost and latency of the proposed method (Figure 12). Our simulations present a fixed allocation model that considers the worst-case requirements for the demands of the microservices (H=0), and the SRA mechanism with a risk threshold R=5% and a gradually increasing prediction time horizon H that ranges from 4 to 16 periods. When the forecasting mechanism is not utilized, services are allocated statically based on worst case assumptions. This results in increased cost due to

overallocation of edge resources and a greater number of reallocations. Also, as the capacity of both the near and far edge resources is limited, this results in the extinction of available edge resources, guiding the remaining applications to the cloud which results in increased latency and thus a deterioration in the offered QoS. When the SRA makes use of the forecasting output it decreases the total cost and the experienced end-to-end latency of the applications assignment by up to 55% and 23% respectively. We also notice that as the time horizon H increases, the SRA mechanism achieves a slightly improved performance.

Finally, we examined the robustness of the proposed SRA mechanism with respect to workload volatility. Particularly, we adjusted the CPU volatility of the microservices' requirements within a fixed topology by adding Gaussian White Noise with a gradually increasing variance to observe how the model responds. As shown in Figure 13, we compare a forecasting model set with risk threshold R=1% against a worst-case estimate model, which allocates a static provision of 2 CPU cores per microservice. The findings reveal that the forecasting model's utilization ratio declines as volatility intensifies, which hints at a decrease in predictive accuracy, consequently leading to more conservative estimates but maintaining the same level of under provisioning events. Contrastingly, the fixed model's performance is less affected by the volatility, maintaining a consistently lower utilization rate. The proposed SRA mechanism demonstrates resilience, surpassing the worst-case model's efficiency and maintaining stable performance even under extreme volatility scenarios. This underscores the SRA model's potential in managing resources in dynamic, uncertain environments.

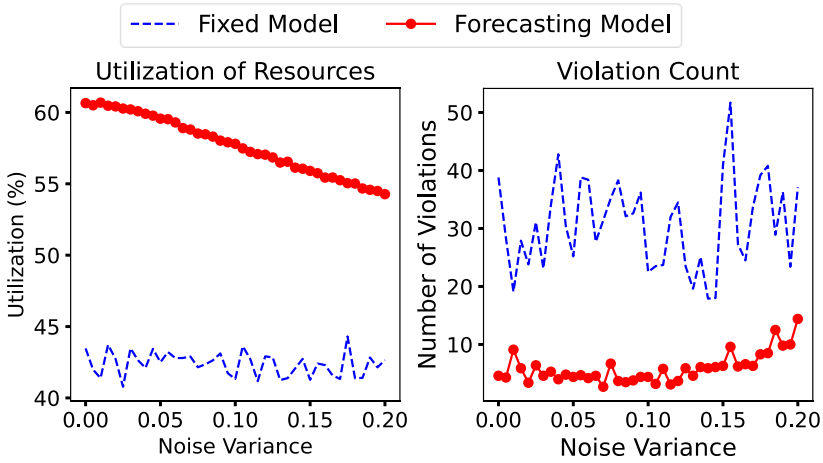


Figure 13: The number of anomalous events and the utilization of allocated resources for different degrees of workload volatility.

5.2.6.3 Conclusion

In this work, we presented a Speculative Resource Allocation (SRA) mechanism, that leverages a stochastic forecasting model drawn on the principles of the Black-Scholes financial model. Designed to address the inherent inefficiencies within edge computing orchestration, characterized by rigid forecasting and disruptive resource scaling, our mechanism performs a speculative risk aware approach. It proactively adjusts to the variable demands of applications, reducing latency and operational costs. The SRA mechanism considers the risk of overprovision events among its objectives, handling the unpredictability of workload

demands. Through this risk-aware approach, it not only anticipates potential demand spikes but also selects the allocation of resources in a way that minimizes the impact of such events. Extensive simulation experiments, based on real-world data, have shown that the proposed mechanism optimizes resource utilization and minimizes service disruption. The results affirm the mechanism's compatibility within the landscape of distributed computing, suggesting a scalable and adaptive solution for the resource orchestration that can competently handle diverse and dynamic, fluctuating workloads with a minimal computational overhead.

5.3 Distributed Knowledge Graphs for the Allocation of Federated Edge-Cloud Resources

5.3.1 Introduction

As the landscape of IoT systems expands, modern applications demand increasingly sophisticated infrastructures that can process large volumes of data in real-time. Traditionally, this processing has relied on centralized cloud resources or simple edge nodes, forming what is commonly referred to as the IoT-edge-cloud continuum. However, the complexity of emerging AI-driven and hyper-distributed applications often exposes the limitations of these monolithic systems, particularly in terms of scalability, privacy, and resource management. To overcome these limitations, we introduce a novel approach based on federations of collaborative, heterogeneous IoT devices and resources, to be referred to as IoT-Edge Associations or simply Associations (Figure 14). These Associations function as federated entities that leverage distributed, AI-driven decision-making to balance computing tasks and optimize resource usage across various providers, networks, and locations.

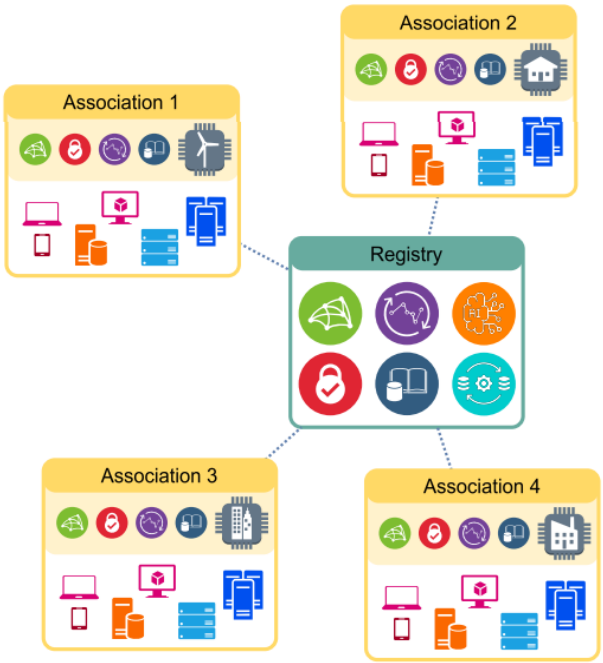


Figure 14: Associations-based continuum.

This work explores an advanced resource allocation framework within such a federated IoT-edge-cloud environment, namely Associations based-continuum, where the interplay between distributed IoT devices, edge nodes and cloud resources are key. Our framework utilizes Distributed Knowledge Graphs (DKGs), a structured approach to formally represent and manage relationships between heterogeneous computational resources, workload tasks, and user entities. Each Knowledge Graph (KG) represents data in a graph-based model, where nodes represent entities (such as resources or jobs), and edges represent relationships between them, in a particular Association. In the context of our framework, DKGs enable each Association to make informed decisions locally, based on aggregated knowledge, while also overcoming the privacy and scalability challenges typically associated with distributed resources by controlling the granularity of information exchanged.

The goal of this work is to address the challenges of optimizing resource allocation in dynamic environments, specifically within federations of Associations, by balancing the trade-offs between execution time, precision, and scalability. Our proposed mechanisms aim to improve collaboration among distributed entities without compromising individual job and resource privacy.

Figure 15 illustrates the overall system architecture, integrating Distributed Knowledge Graphs (DKG) into the Association-based continuum consisting of IoT devices, edge, and cloud resources, user, and provider entities. We assume that a central entity, namely Registry, is responsible for maintaining the DKG. The Registry also hosts the resource allocation strategies proposed in this work that are tailored to federated IoT-edge-cloud environments.

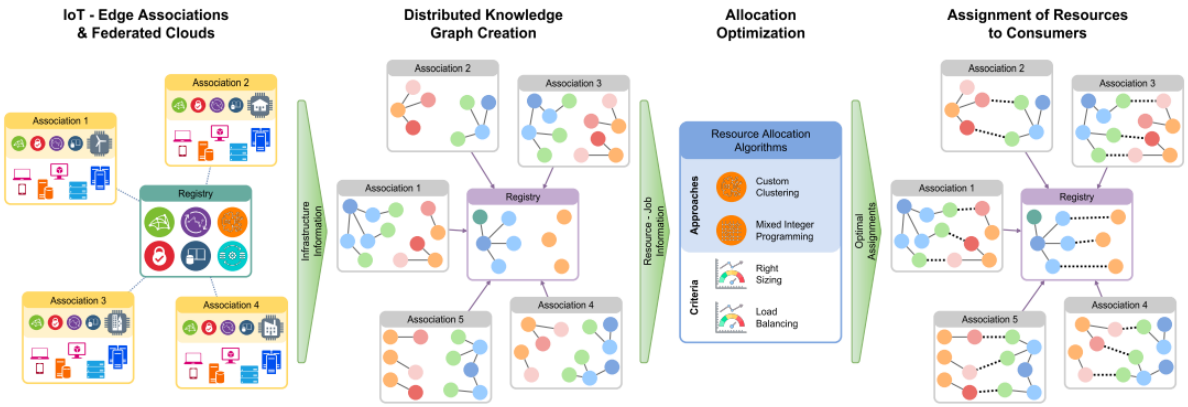


Figure 15: System Architecture: Federated IoT-Edge-Cloud Continuum with DKG for Resource Allocation.

Key research objectives and contributions of this work include:

- A privacy-preserving Distributed Knowledge Graph (DKG) based architecture to manage resource allocation efficiently while maintaining data privacy. This allows resource providers to share only aggregated data without compromising the system's ability to make informed decisions.
- A novel clustering algorithm with a fixed centroids approach for resource allocation. This method is compared with a Mixed Integer Programming (MIP) baseline, offering

insights into the trade-offs between computational efficiency, adaptability, and precision. Our approach is designed to optimize load balancing and right-sizing under different levels of information granularity.

- A comparative evaluation of our method on different information granularity levels. Our study investigates how varying levels of information granularity—ranging from high granularity with detailed job and resource data to reduced granularity with aggregated information—impact resource allocation efficiency and decision speed.

5.3.2 Related Work

The efficient allocation of workload to edge-cloud resources has been addressed using various techniques across different computing paradigms. Integer Linear Programming (ILP) models, multi-objective optimization and metaheuristics techniques have been employed to optimize multiple objectives, such as minimizing server usage while maximizing resource utilization or balancing energy consumption with service level agreements.

Relative recently, Machine learning (ML) techniques have emerged as powerful tools for optimizing resource allocation. ML models predict resource demand, enabling systems to adjust resource distribution dynamically [47][48]. Xiong et al. [49][50] and Qadeer et al. [54] apply deep reinforcement learning (DRL) to resource allocation in mobile edge computing, improving job completion time and resource usage. Recent works have also integrated knowledge graphs with machine learning for advanced resource management, network optimization and event detection Mitropoulou et al. [51][52].

Federated knowledge graphs are promising for managing resources in distributed environments by enabling data integration and sharing across different nodes, facilitating efficient resource allocation and optimization. Federated learning (FL) and federated knowledge graphs (FKG) address challenges of data integration and privacy in distributed systems [53][54]. These approaches allow devices to collaboratively learn a shared model without sharing raw data, preserving privacy while benefiting from collective learning.

Zhu et al. [55][56] discuss the use of federated knowledge graphs for optimizing resource allocation in edge computing environments, highlighting the benefits of data integration and collaboration. Chen and Liu [56] propose a federated deep reinforcement learning-based task offloading and resource allocation method for smart cities, demonstrating the potential of collaborative resource management in complex environments.

In federated computing environments, balancing efficiency with privacy is critical in resource allocation. Techniques such as differential privacy and secure multiparty computation protect data integrity but can introduce computational overhead [57]. Adaptive algorithms that consider both resource constraints and privacy policies enhance system resilience and efficiency, as seen in IoT environments with heterogeneous devices and data sensitivity [58]. Dandoush et al. [59] highlight the importance of decentralized data training without direct data sharing in federated systems, addressing privacy concerns in next-generation networks. Zhan et al. [60] discuss the challenges posed by the diversity of devices in federated networks, which pose significant challenges in resource allocation due to heterogeneity. Sarikaya [61]

explores adaptive strategies that tailor resource allocation based on the specific characteristics of each client, enhancing system efficiency and fairness. Yang et al. [62] address challenges of non-id data distributions in federated environments, impacting model performance in resource allocation. Yu et al. [63] discuss the integration of deep reinforcement learning with federated learning as a promising direction for future research in resource allocation, offering significant potential for improving resource utilization.

Our approach leverages both MIP and custom clustering algorithms to optimize resource allocation within a federated distributed system. By using distributed knowledge graphs, we enhance our ability to integrate and share data across different associations while preserving privacy. Compared to traditional methods, our approach offers enhanced privacy ensuring sensitive data remains decentralized, improved efficiency by balancing trade-offs between objectives, and scalability with the growing number of IoT devices and edge resources.

5.3.3 Problem Formulation

5.3.3.1 Model

The primary objective of our study is to optimize resource allocation in a federated IoT-edge-cloud environment. This consists of collaborative computing ecosystems of heterogeneous IoT devices and resources, from different resource owners (providers), operating in various regions and using various connectivity types. Each such ecosystem, is what we call an Association that realizes an autonomous, secure, and trusted collective. Each Association can adjust the level of information provided to other Associations or third-party entities, regarding its dynamic or static status, in terms of: storage and processing capacities, of resources' specific characteristics (e.g., resource-constrained devices, RISC-V architecture etc), utilization and other parameters of interest.

The envisaged Association-based continuum serves as a bridge between infrastructure and service providers (the supply side) and application developers and end users (the demand side) who require high-performing, low-latency, hyper-distributed applications.

In particular, the key stakeholders considered are the following:

- *Providers*: There are various types of providers: IoT providers, Edge providers and Cloud providers. IoT providers make available IoT infrastructures (e.g., sensors, robots) that generate data. Edge providers offer edge computing and storage resources. Cloud providers provide centralized resources. Service providers also share domain-specific and generic platforms (e.g., for analytics) as a service.
- *Regions*: Regions reflect the distribution of the resources in various locations. For instance: Provider A can have resources in different regions such as 'factory A', 'factory-B', while Provider's B resources can be in 'factor-B' and 'factory-C' etc.
- *Groups*: We further group Resources within each region into groups (e.g., factory-A-1, factory-A-2), indicating particular resource pools. Groups operating in different regions

can be controlled by different resource owners (providers) and used in shared manner by various Users/Customers.

- *Application Developers*: Create hyper-distributed continuum-native applications that are deployed in the continuum and utilize the providers' resources.
- *Users/Consumers*: The end-users are those who interact with and use the hyper-distributed applications.

In practice, these stakeholders can take on multiple roles. For example, an organization can act as both an infrastructure provider and a user. In this dual role, it can contribute a portion of its infrastructure resources to the continuum via an Association, making them available to other users. At the same time, as an application developer and user, the organization can utilize the platform's decentralized intelligence and its application development and deployment solutions to enhance its applications' performance.

Also, this hierarchical resource distribution model consisting of multiple providers, in different regions and groups provides a portrayal of the complexities involved in managing distributed resources across a wide array of locations and administrative domains. Associations collaborate making decentralized decisions to balance the workload intelligently across the decentralized computing environments: inside an Association, between Associations, between Associations and other edge resources or central cloud computing centers.

Our work aims to achieve optimal allocation of computational workloads (jobs) to the resources of each Association, in a way that minimizes allocation inefficiencies while adhering to resource constraints and considering the granularity of available information.

5.3.3.2 Approach

Our pipeline is structured into distinct phases, each crucial for exploring and enhancing resource management strategies within complex federated environments.

- Data Acquisition from Associations: In the initial phase, data from multiple associations are collected to construct the Distributed Knowledge Graph (DKG). This data encompasses information about providers, regions, resources, and users within each Association. The granularity of information shared varies across different scenarios impacts the optimality of any resource allocation algorithm.
- Modelling with Distributed Knowledge Graphs (DKG): This, post data generation, phase involves structuring the synthetic data into a Distributed Knowledge Graph. The DKG encapsulates various entities (e.g., providers, regions, resources, and consumers) and their interrelationships within multiple Associations. Each Association operates autonomously yet collaboratively within the continuum.
- Resource Allocation Optimization: Building on the structured insights provided by the DKG, this phase employs two optimization techniques, developed in this work: Mixed Integer Programming (MIP) and a Custom Clustering (CC) approach. These techniques are assessed for their effectiveness in efficiently allocating resources, based on the granularity of information provided by the DKG.

- Assignment of Resources to Associations: The final stage of the pipeline focuses on the practical application of optimized allocation strategies, where resources are assigned to specific associations. This stage tests the effectiveness of allocation strategies under varying informational scenarios.

5.3.3.3 Modelling with Distributed Knowledge Graphs

Distributed Knowledge Graphs (DKGs) serve as the architectural backbone for modelling the interactions and relationships within the Associations-based continuum. This is a federated system constructed around multiple Associations, each represented as an individual Knowledge Graph (KG).

Each Association's KG encapsulates a subset of the system's resources, users, and operational metadata that vary in type, capacity, and function. As depicted in the metagraph of Figure 16, these resources range from high-performance computing units to simpler storage solutions, all linked via a network that mimics real-world distributed computing environments. The various entities (Providers, Regions, Groups, Resources, Users/Consumers and Jobs) are represented as nodes while their interconnections are represented as edges. The granularity of the data captured within each Association's knowledge graph includes specific attributes, namely CPU cores, GPU cores, memory capacity, and network bandwidth that are represented as node properties. As shown, the KG is a disconnected graph since there are no edges between the resources and the demands. These edges will be created after the allocation process is complete, as a result of our resource allocation algorithms.

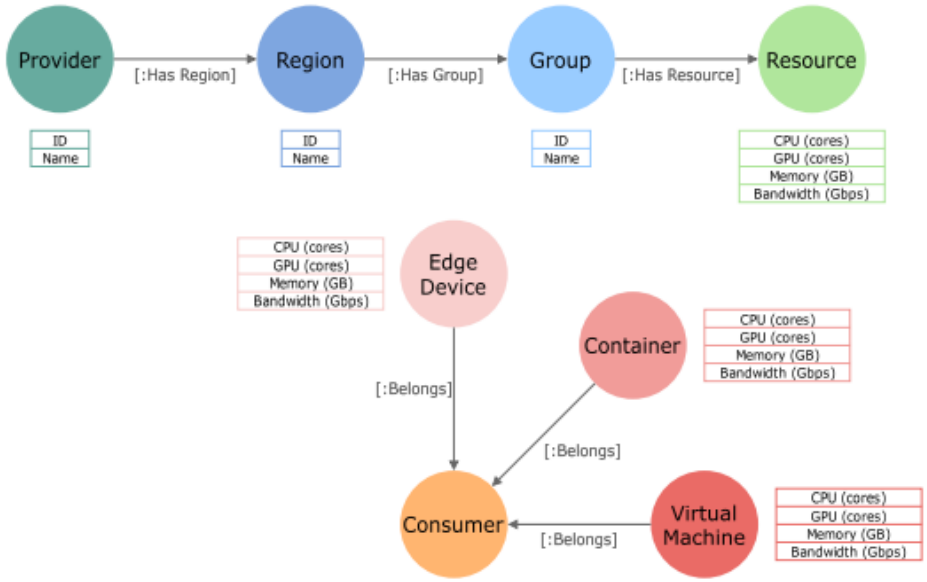


Figure 16: Metagraph of Association Knowledge Graph.

Central to our model is the Registry, a meta-layer that oversees and interacts with the individual Associations. The Registry's role is to compile and synthesize information across different Associations, forming a comprehensive view of the network's resources and demands. The consolidated view offered by this knowledge graph facilitates strategic decision-making processes, particularly in the optimization of resource allocation across the federated system. The Registry can access varying levels of detail from each Association, depending on the granularity of information provided.

5.3.3.4 Scenarios of Information Granularity

The efficacy of resource allocation strategies is significantly influenced by the level of detail available within the Registry's KG. To explore this, we consider two distinct scenarios:

- ***High Granularity Scenario***: In this scenario, each Association provides the Registry with detailed information about individual resources and consumer demands. This includes precise specifications of resource capacities and the specific requirements of each job or process. By sharing this level of detail, the Registry can make highly informed and accurate resource matching and allocation decisions, potentially optimizing utilization and improving efficiency across the network.
- ***Reduced Granularity Scenario***: This scenario restricts the information to aggregated data at the group level within each region managed by the providers. Associations offer summarized details about the resources, such as total CPU and GPU capacities per group, without revealing information about individual units. Similarly, consumer demands are aggregated, reflecting the overall demand per consumer rather than specific job requirements. This scenario evaluates the efficiency of resource allocation under data limitations, simulating conditions where privacy concerns or restricted data availability hinder the flow of detailed operational data.

These scenarios enable the assessment of how varying levels of transparency within a federated system, such as the Association-based continuum, influence the performance and outcomes of resource allocation strategies.

5.3.4 Resource Allocation optimization approaches

5.3.4.1 Resource Allocation Optimization using Mixed Integer Programming (MIP)

Mixed Integer Programming (MIP) is a robust mathematical optimization technique widely employed to solve problems involving both continuous and discrete variables. To address the resource allocation problem within the federated distributed system, we formulated the following MIP model.

The primary objective is to optimize the allocation of heterogeneous resources to jobs and at the same time foster fairness for all involved parties, providers, and consumers.

To this end, we formulate the optimization problem to achieve two main goals:

1. **Right Sizing**: Ensuring that the resources allocated to each job closely match their requirements to minimize wastage and redundant charging.
2. **Load Balancing**: Distributing the load evenly across available resources to avoid overloading any single resource and improve overall system performance.

Binary Decision Variables

As mentioned, the resource allocation problem is translated to an edge creation problem for the disconnected DKG. An edge that connects the job *i* to the resource *j* can be modelled as a binary variable that indicates whether this edge exists. Therefore, the decision variables x_{ij} of the above problem, are defined as follows:

$$x_{ij} = \begin{cases} 1, & \text{if resource } j \text{ is allocated to job } i, \\ 0, & \text{if resource } j \text{ isn't allocated to job } i \end{cases},$$
$$\forall i \in \{1, \ldots, N_{jobs}\}, \forall j \in \{1, \ldots, N_{res}\}.$$

Objective Function

The objective function of the MIP model is a combination of two parts, reflecting the goals of right sizing and load balancing. The purpose is to define penalizing metrics for each one of them and proceed by minimizing the overall penalty:

Right Sizing (RZ): This part of the objective function penalizes the oversupply of resources j with regard to the requirements of each job i. The oversupply of each requirement (CPU, GPU, Memory and Bandwidth) is calculated as a squared percentage relative to the job's demand. By design, there is no prioritization among requirements. Hence, the total oversupply per job is calculated as the sum of the individual oversupplies. GPU requirement has the particularity that it may be equal to zero. We utilize the max operator to prevent division by zero resulting in infinite terms. The following equation depicts the calculation for the entire Association:

$$\begin{aligned} \mathrm{RZ} = \sum_{i=1}^{N_{jobs}} \sum_{j=1}^{N_{res}} x_{ij} \cdot \Bigg(& \left(\frac{CPU_j - CPU_i}{CPU_i} \right)^2 + \\ & \left(\frac{\max(GPU_j, 1) - \max(GPU_i, 1)}{\max(GPU_i, 1)} \right)^2 + \\ & \left(\frac{Memory_j - Memory_i}{Memory_i} \right)^2 + \\ & \left(\frac{Net.Band._j - Net.Band._i}{Net.Band._i} \right)^2 \Bigg) \end{aligned}$$

Load Balancing (LB): This part ensures that the workload is balanced across the resources, aiming for an even percentage utilization relative to an overall system balance parameter for each requirement. The system balance is defined as the total demand by the total supply within the Association. In order to achieve overall balance, we need to ensure that the resource utilization follows a uniform distribution that has a mean value equal to the system balance. For each resource we calculate and penalize the squared difference between its percentage utilization and the system balance. Similarly to the above part, there is no prioritization among requirements, hence, the total difference is calculated as the sum of the individual differences. Additionally, we utilize the max operator to prevent division by zero in GPU related terms.

The following equations depict these calculations:

$$\begin{aligned}\text{LB} = \sum_{i=1}^{N_{jobs}} \sum_{j=1}^{N_{res}} x_{ij} \cdot \Bigg(& \left(\frac{CPU_i}{CPU_j} - \text{System CPU Balance} \right)^2 + \\ & \left(\frac{\max(GPU_i, 1)}{\max(GPU_j, 1)} - \text{Sys. GPU Balance} \right)^2 + \\ & \left(\frac{Memory_i}{Memory_j} - \text{Sys. Memory Balance} \right)^2 + \\ & \left(\frac{\text{Net. Band.}_i}{\text{Net. Band.}_j} - \text{Sys. Net. Band. Balance} \right)^2 \Bigg)\end{aligned}$$

,where:

$$\text{System CPU Balance} = \frac{\sum_{i=1}^{N_{jobs}} CPU_i}{\sum_{j=1}^{N_{res}} CPU_j}$$

$$\text{System GPU Balance} = \frac{\sum_{i=1}^{N_{jobs}} GPU_i}{\sum_{j=1}^{N_{res}} GPU_j}$$

$$\text{System Memory Balance} = \frac{\sum_{i=1}^{N_{jobs}} Memory_i}{\sum_{j=1}^{N_{res}} Memory_j}$$

$$\text{System Net. Band. Balance} = \frac{\sum_{i=1}^{N_{jobs}} Net.Band._i}{\sum_{j=1}^{N_{res}} Net.Band._j}$$

Constraints

The model incorporates several constraints to ensure that the resource allocation is feasible and respects the capacity limits of the resources: Resource Capacity Constraints, ensure that the total jobs served by each resource do not exceed its available capacities:

$$\sum_{i=1}^{N_{jobs}} x_{ij} \cdot CPU_i \leq CPU_j, \quad \forall j \in \{1, \ldots, N_{res}\}$$

$$\sum_{i=1}^{N_{jobs}} x_{ij} \cdot GPU_i \leq GPU_j, \quad \forall j \in \{1, \ldots, N_{res}\}$$

$$\sum_{i=1}^{N_{jobs}} x_{ij} \cdot Memory_i \leq Memory_j, \quad \forall j \in \{1, \ldots, N_{res}\}$$

$$\sum_{i=1}^{N_{jobs}} x_{ij} \cdot Net.Band._i \leq Net.Band._j, \quad \forall j \in \{1, \ldots, N_{res}\}$$

Allocation Constraints, ensure that each resource is allocated to exactly one job:

$$\sum_{i=1}^{N_{jobs}} x_{ij} \leq 1, \quad \forall j \in \{1, \ldots, N_{res}\}$$

Serving Constraints, ensure that each job is served by at least one resource:

$$\sum_{j=1}^{N_{res}} x_{ij} \geq 1, \quad \forall i \in \{1, \ldots, N_{jobs}\}$$

Optimization Model

The Mixed Integer Programming (MIP) model:

- Minimize Right Sizing and Load Balancing
- subject to:
    - Resource Capacity Constraints
    - Allocation Constraints
    - Serving Constraints

MIP Algorithm

The following algorithm provides a detailed outline of the implementation of the MIP model for resource allocation:

Algorithm 1 Resource Allocation Optimization - MIP Algorithm

Require: N_{res}, N_{jobs}
$CPU_i, GPU_i, Memory_i, Net\_Band_i$
$CPU_j, GPU_j, Memory_j, Net\_Band_j$
Ensure: Optimal allocation of jobs to resources

Define MIP model
$model.initialize$

// Define binary decision variables
for $j \leftarrow 1$ **to** N_{rec}, $i \leftarrow 1$ **to** N_{jobs} **do**
    $model.variables \leftarrow$ Add $x_{ij} \in \{0, 1\}$
end for

// Define objective function
$model.objective \leftarrow$ Right Sizing (Eq. 2) + Load Balancing (Eq. 3)

// Define Constraints
for $j \leftarrow 1$ **to** N_{rec} **do**
    $model.constraints \leftarrow$ Add Resource Capacity Constraints (Eq. 5-8)
    $model.constraints \leftarrow$ Add Allocation Constraints (Eq. 9)
end for

for $i \leftarrow 1$ **to** N_{jobs} **do**
    $model.constraints \leftarrow$ Add Serving Constraints (Eq. 10)
end for

Solve MIP model
$model.optimize$

5.3.4.2 Custom Clustering Approach with Fixed Centroids

As an alternative to the Mixed Integer Programming (MIP) approach, we developed a Custom Clustering method to optimize resource allocation. This approach aims to combat the convergence time of MIP by materializing a clustering model with fixed centroids and a custom distance metric. The primary goal is to efficiently assign jobs to resource pools while respecting the resource constraints and ensuring a balanced load distribution.

Similarly to the MIP approach, we aim at optimizing two elements, namely the Right Sizing and the Load Balancing. The goal is to minimize the overall discrepancy of the federated system, which is equivalent to the cumulative discrepancies produced by each allocation. In the scope of a clustering task, where the purpose is to minimize the distance between the points to their assigned centroids, this is translated to minimize the distance from the optimal allocation, which ultimately results in minimizing the cumulative distance within the set of points.

Consequently, the distance in our custom clustering approach is equivalent to the objective function from MIP, but without the summation over all allocations. Instead, we calculate a custom metric that comprises the discrepancy as the sum of the two penalties (Right Sizing and Load Balancing), between the points (jobs) and the centroids (resource types) and aim at minimizing it. To incorporate the existing constraints into our custom distance metric, we add

an additional penalty term of a greater order of magnitude as described in the Penalty equation. This ensures that infeasible allocations are heavily penalized, thereby preventing them from occurring.

The summation occurs iteratively during the clustering process, where in each iteration, jobs (points) are assigned to the nearest resource pools (centroids), and the discrepancies are computed and minimized iteratively. This iterative process continues until the assignments are stable and the total discrepancy is minimized.

Another concept we introduce in our custom clustering approach is the cluster size. This means that every cluster can comprise up to a specific number of points. Since every centroid corresponds to a resource type available in the federated system, and each resource type has a fixed number of instances, each cluster has an upper limit on the number of points it can include. In other words, this means there is a maximum number of jobs that each resource type can handle.

The core idea of our custom clustering algorithm involves the following steps, depicted in more detail in Algorithm 2:

1) Data Preparation: Jobs are represented in space by their resource requirements (CPU, GPU, memory, bandwidth). Resource pools are represented by their capacities and number of occurrences within the federated system, which serve as fixed centroids and cluster sizes respectively.
2) System Balance Calculation: We compute the total demand and capacity for each resource type (CPU, GPU, memory, bandwidth) across all jobs and resource pools. The system balance is calculated by dividing the total demand by the total capacity for each resource type.
3) Clustering Algorithm:
    - Initialization: The algorithm starts with fixed centroids that represent the capacities of the resource pools. Each centroid is defined by its resource capacities, including CPU, GPU, memory, and bandwidth. The system balance was calculated as the ratio of total demand to total capacity for each resource type.
    - Assignment: Each job is assigned to the nearest resource pool (centroid) based on the custom distance metric we describe above that incorporates the penalty term for the constraint handling. The distance metric used for this assignment is:

$$\text{Distance} = \left(\frac{CPU_j - CPU_i}{CPU_i}\right)^2 + \left(\frac{\max(GPU_j,1) - \max(GPU_i,1)}{\max(GPU_i,1)}\right)^2 + \left(\frac{Memory_j - Memory_i}{Memory_i}\right)^2 + \left(\frac{Net.Band._j - Net.Band._i}{Net.Band._i}\right)^2 + \left(\frac{CPU_i}{CPU_j} - \text{Sys. CPU Balance}\right)^2 + \left(\frac{\max(GPU_i,1)}{\max(GPU_j,1)} - \text{Sys. GPU Balance}\right)^2 + \left(\frac{Memory_i}{Memory_j} - \text{Sys. Memory Balance}\right)^2 + \left(\frac{Net.Band._i}{Net.Band._j} - \text{Sys. Net. Band. Balance}\right)^2 + \text{Penalty}(i,j),$$

$$\text{Penalty}(i,j) = \begin{cases} 10^6, & CPU_i > CPU_j \text{ or} \\ & GPU_i > GPU_j \text{ or} \\ & Memory_i > Memory_j \text{ or} \\ & Net.Band._i > Net.Band._j \\ 0, & otherwise \end{cases},$$

$$\forall i \in \{1,\ldots,N_{jobs}\}, \forall j \in \{1,\ldots,N_{res}\}$$

$$\text{System CPU Balance} = \frac{\sum_{i=1}^{N_{jobs}} CPU_i}{\sum_{j=1}^{N_{res}} CPU_j}$$

$$\text{System GPU Balance} = \frac{\sum_{i=1}^{N_{jobs}} GPU_i}{\sum_{j=1}^{N_{res}} GPU_j}$$

$$\text{System Memory Balance} = \frac{\sum_{i=1}^{N_{jobs}} Memory_i}{\sum_{j=1}^{N_{res}} Memory_j}$$

$$\text{System Net. Band. Balance} = \frac{\sum_{i=1}^{N_{jobs}} Net.Band._i}{\sum_{j=1}^{N_{res}} Net.Band._j}$$

$$\forall i \in \{1,\ldots,N_{jobs}\}, \forall j \in \{1,\ldots,N_{res}\} \quad (12)$$

- Iterative Adjustment: Unlike traditional K-Means, where centroids are updated iteratively, our centroids remain fixed. Instead, the assignment of jobs to centroids is iteratively adjusted to minimize the total discrepancy with regard to the two goals explained in Problem Formulation. The algorithm re-evaluates assignments in each iteration, reassigning jobs from over-allocated pools to those with available capacity. The process continued until the loss function converged to a predefined tolerance level or a maximum number of iterations was reached.
- Handling of Constraints: Constraints are managed through the penalty within the custom distance metric. Distance imposes significant penalties for assignments exceeding resource capacities. This ensures that resource allocations remain within feasible limits.

4) Cluster Size Handling: The algorithm ensures that the number of points assigned to each centroid does not exceed the predefined cluster size. If a resource pool is over-allocated, the algorithm reassigns jobs to other pools, balancing the load while respecting the threshold of each cluster.

5) Output: The final output is the assignment of jobs to resource pools, ensuring that each job is allocated to a resource pool that can meet its requirements without exceeding the pool's capacities.

The following algorithm provides a detailed outline of the implementation of the Custom Clustering based mechanism for resource allocation:

Algorithm 2 Resource Allocation Optimization: Custom Clustering Algorithm

Require: $N_{res}, N_{jobs}, Centroids_{res}, ClusterSizes_{res}$
$CPU_i, GPU_i, Memory_i, Net\_Band_i$
$CPU_j, GPU_j, Memory_j, Net\_Band_j$
max_iter

Ensure: Optimal allocation of jobs to resources

Define Custom Clustering model
Class $CustomClustering\{$

// Set parameters: Resource pools, jobs, cluster size
def $initialize(N_{res}, N_{jobs}, Centroids_{res}, ClusterSizes_{re}$
$CPU_i, GPU_i, Memory_i, Net\_Band_i,$
$CPU_j, GPU_j, Memory_j, Net\_Band_j)$

// Set distance metric: as defined in Eq. 12
def $distance(CPU_i, GPU_i, Memory_i, Net\_Band_i,$
$CPU_j, GPU_j, Memory_j, Net\_Band_j,$
$Penalty = 10^6)$

// Set loss: Sum of distances
def $evaluate\_loss(distance)$

// Assign jobs to centroids based on distance minimization
while respecting cluster size
def $assign\_jobs\_to\_centroids(Centroids_{res},$
$ClusterSizes_{res}, distance)$

// Fit iteratively until maximum iteration is reached
def $fit(assign\_jobs\_to\_centroids, evaluate\_loss,$
$max\_iter)$

$\}$

Execute Optimization
// Create object of class CustomClustering
$model \leftarrow$ CustomClustering$(N_{res}, N_{jobs}, Centroids_{res},$
$ClusterSizes_{res}, CPU_i, GPU_i, Memory_i, Net\_Ban$
$CPU_j, GPU_j, Memory_j, Net\_Band_j)$

// Fit Custom Clustering model
$model.fit$

5.3.4.3 Comparison of Mixed Integer Programming with Custom Clustering with Fixed Centroids

Both the Mixed Integer Programming (MIP) and Custom Clustering approaches aim to solve the resource allocation optimization problem efficiently, but they do so from different perspectives.

The MIP approach formulates the resource allocation problem as a mathematical optimization problem involving both continuous and discrete variables. This method systematically evaluates each possible allocation, ensuring that resource utilization is maximized and the load is evenly distributed across available resources. However, the computational complexity and convergence time of MIP can be significant, especially for large-scale problems.

In contrast, the Custom Clustering approach simplifies the problem by using fixed centroids, representing resource pools with predefined capacities. This method clusters jobs based on their resource requirements, assigning them to the most suitable resource pools. The fixed centroids guide the assignment process, while the iterative process reduces the complexity and improves the scalability of the solution. While this approach may not guarantee an optimal solution like MIP, it provides a more efficient method for resource allocation, particularly suitable for scenarios with stable resource characteristics.

The objective functions in both approaches aim to achieve right sizing and load balancing but are tailored to their respective methodologies. In the MIP approach, the objective function is defined with a comprehensive exploration of all possibilities, ensuring the best possible resource utilization and load balancing. The Custom Clustering approach adapts the objective function to define the distance metric used to fit the clustering model, focusing on minimizing the discrepancy between job requirements and resource capacities while ensuring balanced load distribution.

5.3.5 Performance Evaluation

This section presents the performance evaluation of the two optimization algorithms developed: Mixed Integer Programming (MIP) and Custom Clustering. Our simulation includes multiple associations, resource types, and consumer job profiles, designed to mirror real-world distributed computing environments (Table 5). These variations in size allow us to explore how each method scales and adapts under increasing complexity and load. The experiments examine key performance metrics: consumers served, load balancing, right sizing, and execution time. We consider both high granularity and reduced granularity scenarios to understand how each algorithm performs under varying levels of information availability.

Table 5: Simulation Sizes.

Simulation Load	Consumers	Jobs	Groups	Resources
1	125	375	125	600
2	137	412	137	660
3	150	450	150	720
4	162	487	162	780
5	175	525	175	840
6	187	562	187	900
7	200	600	200	960
8	212	637	212	1020
9	225	675	225	1080
10	237	712	237	1140
11	250	750	250	1200

Figure 17 displays the percentage of consumers served using MIP and Custom Clustering. In the high granularity scenario, MIP consistently manages to serve all consumers across all tested simulation sizes. The availability of detailed job-specific data allows MIP to allocate resources optimally, fully matching job demands to the available resource capacities. Like MIP, Custom Clustering achieves 100% consumer coverage in the high granularity scenario across all simulation sizes as it offers a feasible solution, leveraging the availability of detailed information.

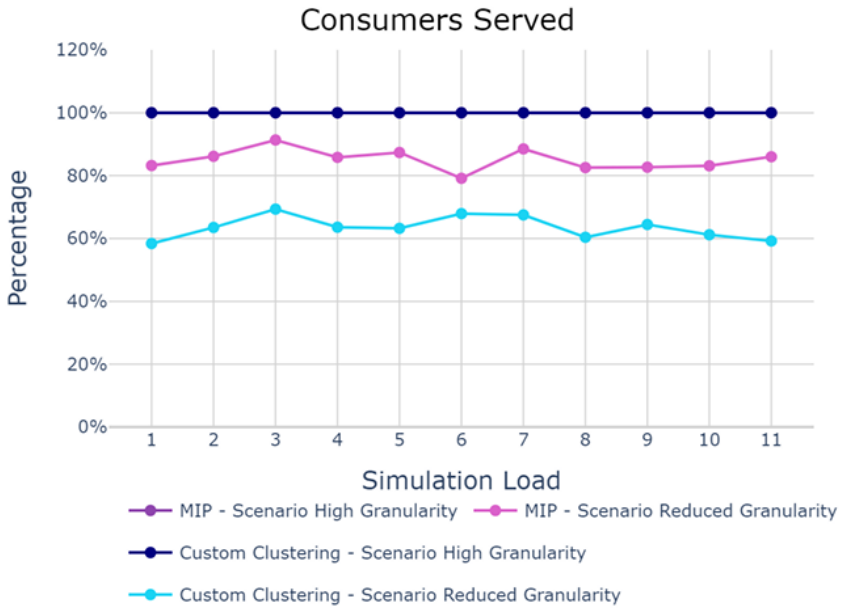


Figure 17: Comparison of consumers served between different scenarios under the MIP and the Custom Clustering algorithms.

In contrast to the high granularity use case, the reduced granularity scenario presents significant challenges. Here, Custom Clustering is unable to serve all consumers, as aggregated information limits the algorithm's visibility into individual job requirements. As expected, this leads to inefficiencies in resource allocation. The reduction in served consumers is not affected by larger simulation sizes, fluctuating between 80% and 90% respectively. On the other hand, the MIP approach shows greater resilience than Custom Clustering in reduced granularity settings. While it also experiences a decrease in the number of consumers served, the drop is less pronounced compared to Custom Clustering. This robustness stems from its inherent flexibility in grouping jobs and resources, even with partial information, allowing it to maintain relatively high levels of job fulfillment and consumer satisfaction.

MIP achieves more precise right-sizing by closely matching resource allocations to the specific needs of individual jobs than Custom Clustering (Figure 18). This is evident in the lower right-sizing penalties observed in the experiments, in both high and low granularity scenarios.

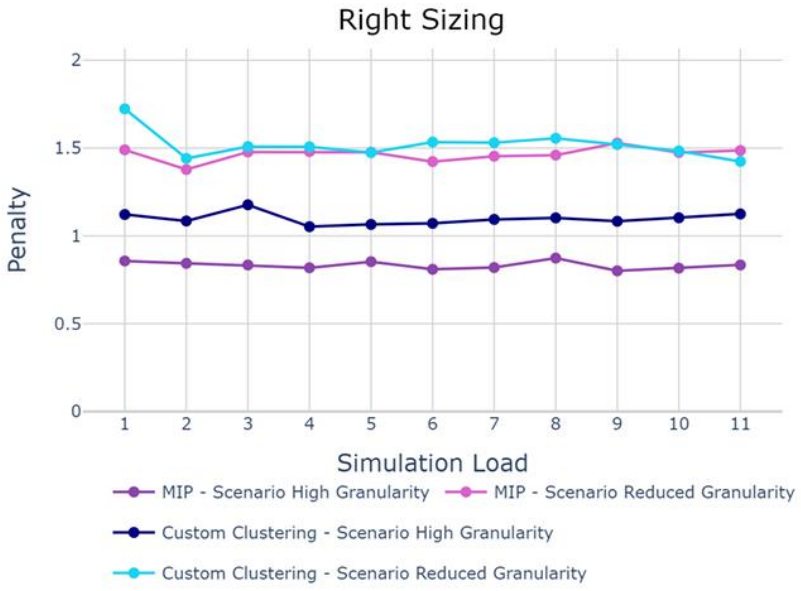


Figure 18: Comparison of Right Sizing penalty between different scenarios under the MIP and the Custom Clustering algorithms.

On the other hand, Custom Clustering shows a remarkable ability to achieve better load balancing under the same high granularity conditions (Figure 19). Despite not being designed to find the absolute optimal allocation, it manages to distribute workloads across resources with efficiency, as reflected in the consistently lower load balancing penalties compared to MIP. This flexibility and adaptability give Custom Clustering an edge in scenarios where minimizing computational overhead and execution time are more critical than achieving the absolute minimum in allocation penalties.

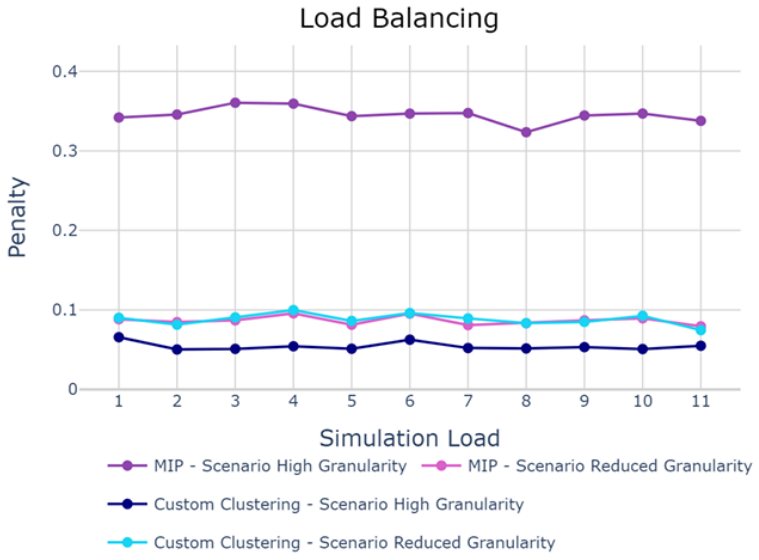


Figure 19: Comparison of Load Balancing penalty between different scenarios under the MIP and the Custom Clustering algorithms.

Figure 20 reveals that MIP's execution time is significantly longer, especially as the simulation size increases. While Custom Clustering is, by definition, less optimal in terms of resource allocation, its much faster execution times make it a more practical option for real-time or time-sensitive applications where decision-making speed is essential.

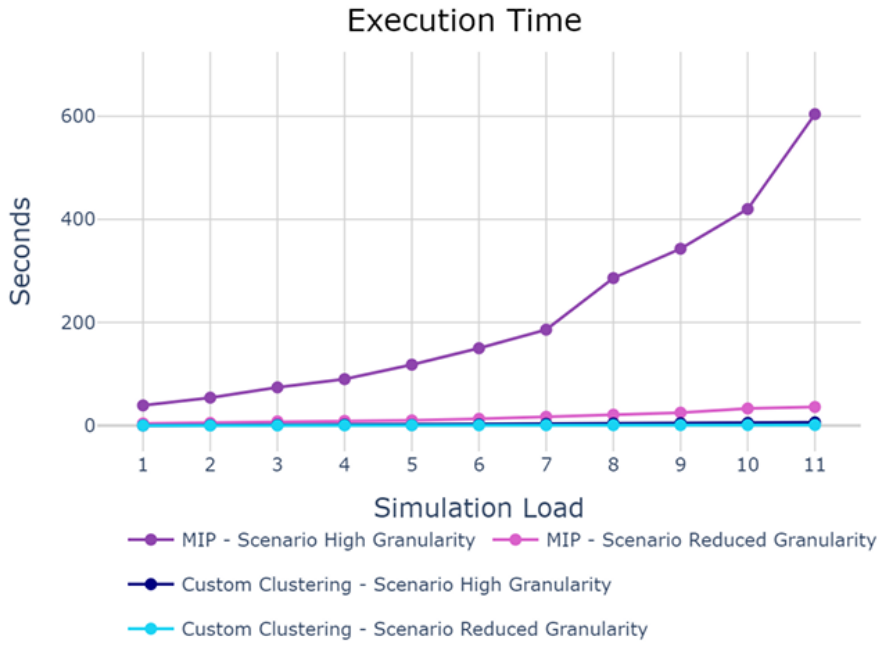


Figure 20: Comparison of execution time between different granularity scenarios, under the MIP and the Custom Clustering algorithms.

6 Decision Engine

6.1 Overview

The management of infrastructures across the IoT-edge-cloud continuum and deployment of cloud-native applications usually rely on hierarchical orchestration models. In these systems, a high-level orchestrator assigns application microservices and workloads to local orchestrators, which, in turn, control subsets of infrastructure resources. Although this approach has been effective, it follows a monolithic paradigm inadequate for the next generation of hyper-distributed and AI/ML-based applications. These applications demand real-time adaptability, low latency, and stringent performance guarantees.

Instead of relying on a rigid, centralized hierarchy, EMPYREAN's Decision Engine leverages an Association-based model that promotes local decision-making and collaborative intelligence. This approach provides cognitive, AI-driven optimizations and autonomous multi-agent coordination towards a self-adaptive and federated continuum. As presented in D2.3 (M12), the deployment of cloud-native applications within the EMPYREAN platform is structured into three distinct phases. The first phase involves decentralized and speculative resource orchestration, where application workloads are strategically distributed to specific Associations according to deployment objectives. The second phase encompasses hierarchical cognitive resource orchestration within the selected Associations, further refining the deployment plan based on platform-specific deployment objectives. The third phase involves the selection of specific worker nodes across the utilized clusters and the actual deployment.

The Decision Engine plays a crucial role in the first two key phases, acting as the intelligence layer that enables EMPYREAN to seamlessly orchestrate hyper-distributed applications, ensuring scalability, efficiency, and sustainability across the edge-cloud continuum. According to the EMPYREAN architecture, the Decision Engine supports the EMPYREAN Aggregator and Service Orchestrator to (i) orchestrate the hyper-distributed applications and distribute their workloads considering the local resource state and characteristics while trying to fulfil their objectives, (ii) coordinate the efficient load-balancing of data and workload within and across the available Associations, and (iii) support the efficient operation of Associations.

6.2 Relation to EMPYREAN Objectives and KPIs

The Decision Engine is one of EMPYREAN's enabling technologies that, in cooperation with the EMPYREAN Aggregator and Service Orchestrator, support efficient and cognitive orchestration and workload placement across the Association-based continuum. To this end, the Decision Engine contributes to the achievement of the following key objectives and technical KPIs[2]:

[2] Technical KPI identifiers introduced in D2.3 (M12).

- *T1.1 - Reduce cloud and increase edge utilization via workload balancing optimization*: Decision Engine enables decentralized and multi-agent orchestration, dynamically guiding the distribution of application workloads across edge and cloud resources, reducing reliance on centralized cloud computing. Its speculative resource allocation and AI-driven heuristics will optimize workload placement closer to data sources, minimizing latency and unnecessary cloud usage.
- *T2.1 - Improve overall performance compared to SotA*: The Decision Engine leverages AI-enhanced heuristics, game theory, and reinforcement learning to optimize execution strategies beyond traditional orchestration solutions.
- *T2.2 - Reduce energy consumption on Associations compared to standard execution*: The Decision Engine incorporates energy constraints into its optimization models, prioritizing workload execution on low-power and renewable-powered devices. It will provide energy-aware scheduling and load redistribution for energy efficiency, favouring green resources to minimize overall carbon footprint.
- T2.4 - Boost AI-driven decision-making accuracy: The Decision Engine supports cooperative and competitive AI agents, allowing for dynamic, context-aware decision-making. It also leverages historical data and reinforcement learning to enhance decision accuracy over time, surpassing rule-based orchestration systems.

6.3 Architecture

The Decision Engine is designed to function seamlessly across diverse computing environments, ranging from edge devices to central cloud infrastructures. By integrating AI-driven, decentralized orchestration, EMPYREAN ensures system-wide welfare optimality, enabling a more efficient, resilient, and scalable computing paradigm for next-generation IoT, AI, and hyper-distributed applications.

Within the EMPYREAN project, efforts focus on extending the open-source Resource Optimization Toolkit (ROT), initially developed in the H2020 SERRANO[3] EU project by the ICCS, into the cloud-native Decision Engine component. As a cloud-native and scalable service, the Decision Engine is specifically designed and built to support the EMPYREAN Multi-Cluster Orchestration layer's cognitive and distributed orchestration requirements. It also integrates a wide range of decision-making algorithms.

As illustrated in Figure 21, the Decision Engine architecture includes a *Decision Engine Controller* and multiple *Execution Engines*, which function as the actual workers. Each worker consists of the Execution Engine and the library of decision algorithms, ensuring both flexibility and scalability. This modular design enables the Decision Engine to efficiently process a high volume of execution requests from the Service Orchestrator, even in highly complex infrastructures.

[3] https://ict-serrano.eu

The Decision Engine interacts directly with the Service Orchestrator, processing service requests while leveraging real-time telemetry data from the EMPYREAN Telemetry Service to make informed, adaptive decisions. Additionally, it utilizes the Distributed Data Broker service, which leveraging Eclipse Zenoh's[4] capabilities, providing a scalable and dynamic communication environment to support multi-agent operations. This environment supports efficient information sharing, helping agents to coordinate, exchange insights, and collaboratively make decisions as they work towards a common optimization goal.

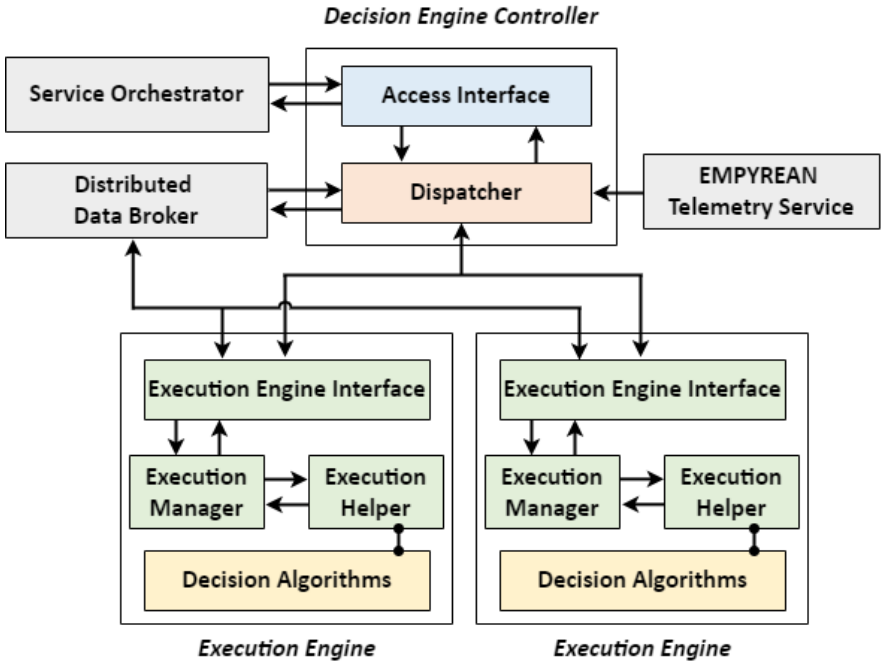


Figure 21: Decision Engine architecture.

The Decision Engine Controller provides bidirectional communication interfaces, enabling the exchange of requests, responses, information, and notifications. To ensure loose coupling between the Decision Engine and other EMPYREAN services, the *Access Interface* exposes RESTful and asynchronous APIs that provide flexible and scalable integration. A detailed description of the provided methods is available in the next section.

At the core of the Decision Engine Controller lies the *Dispatcher*, which provides intelligent task coordination for managing execution requests and facilitating interactions with multiple instances of the Execution Engine. Key functionalities of the Dispatcher include (i) processing and adapting incoming service requests, ensuring alignment with the Decision Engine's execution model, (ii) retrieving real-time telemetry data, including resource characteristics, system status, and application deployments, (iii) creating and managing the communication channels for the multi-agent operation, and (iv) preparing execution parameters and efficiently distributing requests to available Execution Engines for processing.

[4] https://zenoh.io

The Execution Engine processes service requests dispatched by the Decision Engine Controller. Each request is handled as a discrete task that can involve either starting a new execution, including all required input parameters, or terminating an ongoing execution.

The *Execution Engine Interface* first validates incoming service requests from the Dispatcher before forwarding them to the Execution Manager, which handles the actual execution lifecycle. The *Execution Manager* determines whether to terminate an execution or initialize a new one with the assistance of the Execution Helper. The *Execution Helper* focuses on task execution and monitoring by: (i) setting up the execution environment, ensuring all dependencies and configurations are in place, (ii) tracking execution progress and reporting real-time updates to the Execution Manager, and (iii) handling execution results and managing potential failures.

The *Decision Algorithms* is the library of multi-objective optimization and orchestration algorithms. These algorithms drive intelligent decision-making by dynamically balancing trade-offs between performance, resource utilization, energy efficiency, and latency constraints. By leveraging advanced AI-driven heuristics, game theory, and real-time analytics, the Decision Engine continuously refines execution strategies to enhance efficiency, scalability, and adaptability within the EMPYREAN platform.

The updated Decision Engine design introduces capabilities, enabling its evolution into a multi-agent system that significantly improves flexibility, scalability, and decision-making efficiency within the distributed, Association-based continuum of the EMPYREAN platform. By adopting a multi-agent architecture (Figure 22), the Decision Engine supports both cooperative and competitive operations, enabling decentralized and speculative resource orchestration across Associations. This advancement allows multiple autonomous agents, each representing an independent Decision Engine instance orchestrating specific clusters within an Association, to collaborate dynamically for optimizing decision-making.

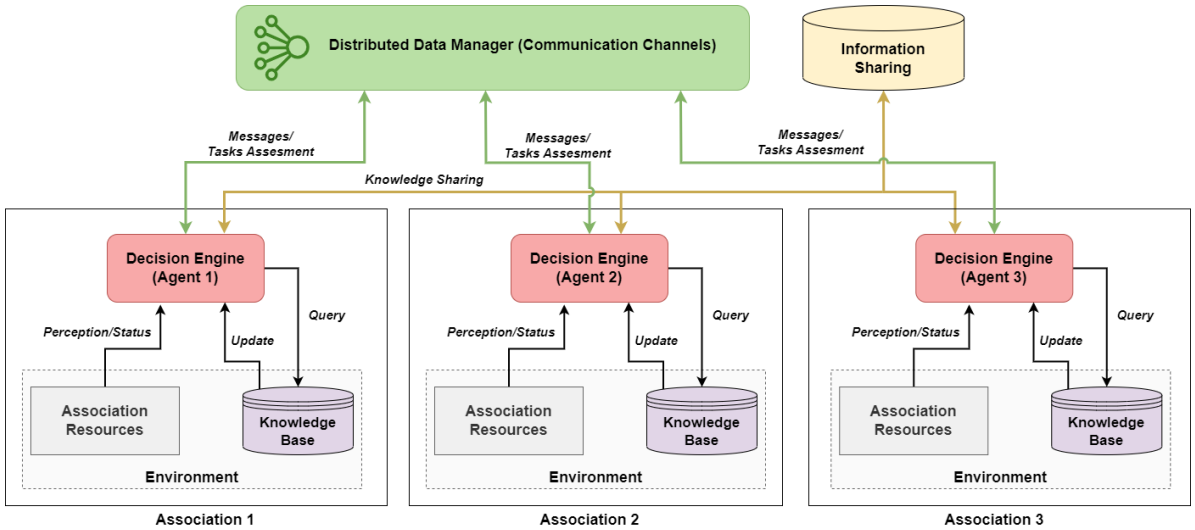


Figure 22: Multi-agent system architecture utilizing multiples instances of Decision Engines across EMPYREAN Associations.

Agents collaborate to optimize workload placement and resource allocation, ensuring efficient use of heterogeneous computing resources across the edge-cloud continuum. By sharing telemetry insights and resource status information, agents collectively negotiate the best possible execution strategies while minimizing bottlenecks and balancing loads across federated Associations. When necessary, agents compete for limited resources, dynamically adjusting their strategies based on game-theoretic principles to ensure fair and efficient resource utilization. Agents use reinforcement learning and adaptive heuristics to continuously improve decision-making based on past performance, real-time demand, and system constraints.

One of the significant aspects of multi-agent decision-making is the ability of agents to convey information and synchronize their actions through communication. Adequate communication is a prerequisite for group work, synchronization, and conflict resolution of agents. To this end, the updated design of the Decision Engine incorporates a lightweight and scalable communication layer based on the Eclipse Zenoh. This middleware provides the necessary mechanisms for inter-agent communication, facilitating the exchange of state information, coordination signals, and messages between Decision Engine instances.

6.4 Implementation

The Decision Engine components are implemented in Python language and packaged in container images, ensuring a smooth and efficient development workflow. There are separate container images for the Decision Engine Controller and Execution Engine.

The Access Interface component provides two Northbound Interfaces (NBIs): one based on REST APIs and the other on an asynchronous messaging interface using the Advanced Message Queuing Protocol (AMQP). The first interface (Figure 23) offers control operations for managing and inspecting the execution of deployment algorithms, retrieving information about the available Execution Engines, and administering end users. The second interface enables asynchronous communication between the Decision Engine Controller and end users, facilitating the exchange of notification messages and results.

The asynchronous interface uses a predefined set of topics for exchanging messages that correspond to various notifications related to the operation of the Decision Engine. These messages are described in JavaScript Object Notation (JSON) format using the following predefined syntax: (a) ***event*** (*string*): event unique identifier and (b) ***data*** (*element*): a set of event-related parameters that provide the required information.

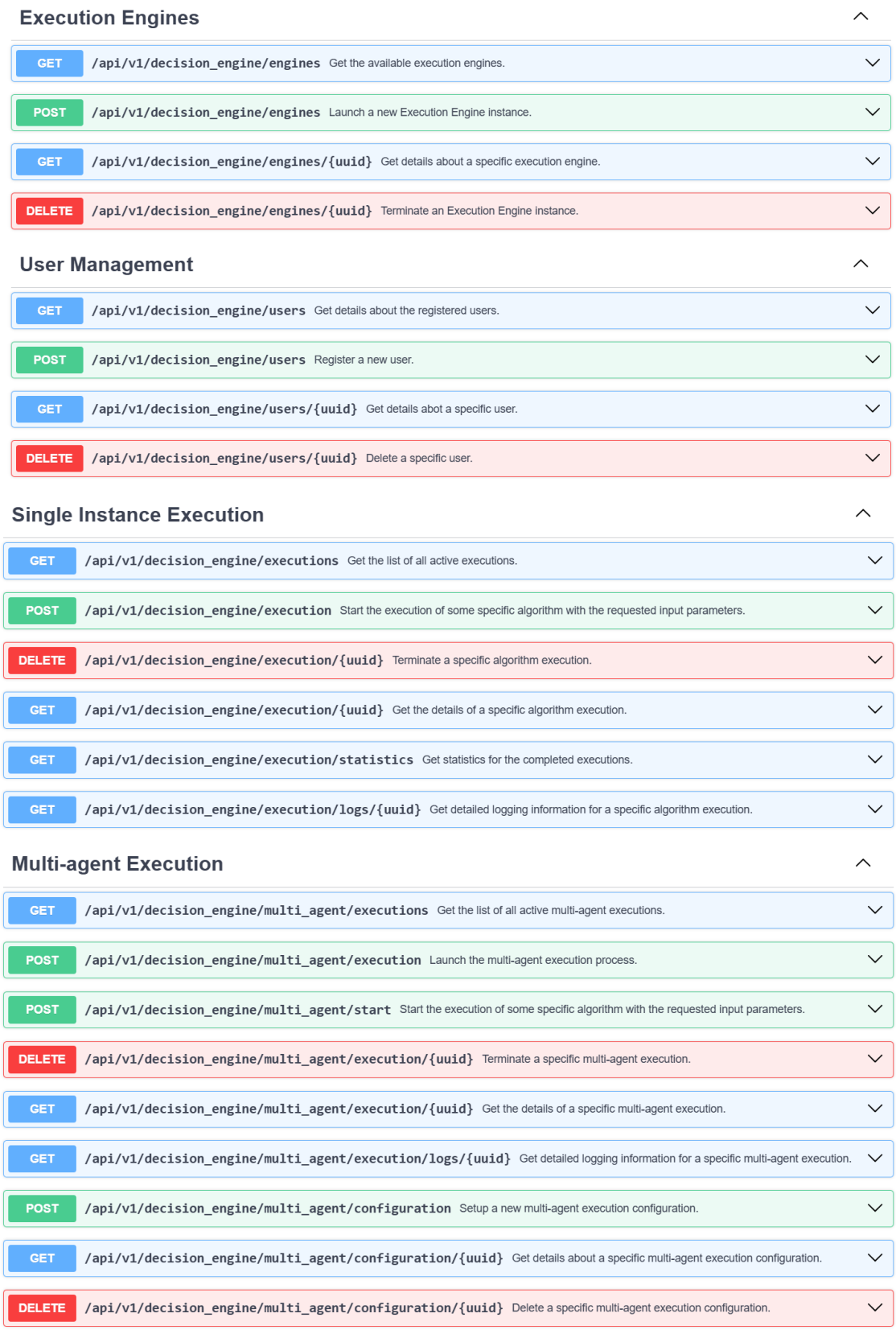


Figure 23: Decision Engine – Access Interface REST API.

Moreover, the workload assignment algorithms are accessible from the Execution Engine internal components through a custom plug-in mechanism. This mechanism exposes a common interface, independent of the implementation technology and algorithm internal logic, that, among others, will determine the explicit syntax of the input parameters and the results for all algorithms. In addition, the common interface uses the JSON as a data-interchange format for providing the input and output parameters.

Next, we provide a detailed workflow description (Figure 24) outlining the multi-agent operation across EMPYREAN Associations, highlighting the roles of the Decision Engine components and their interactions with other services.

1. When the Service Orchestrator needs to determine the initial placement of an application's microservices within the EMPYREAN platform, it sends a service request to its associated Decision Engine (DE) via the Access Interface. This request (*POST /api/v1/decision_engine/multi_agent/execution*) initiates the multi-agent decision-making process. Additionally, the Decision Engine in the originating Association acts as the supervisor during the multi-agent operation.
2. Through the Dispatcher component, the supervisor Decision Engine queries the EMPYREAN Registry to identify Associations that meet the required deployment criteria, utilizing the Registry's exposed RESTful API (Section 11.4).
3. The Decision Engine dynamically creates exchange topics within the Distributed Data Broker to facilitate bidirectional communication among the participating Decision Engines.
4. The supervisor Decision Engine notifies the selected Decision Engines for the created topics (*POST /api/v1/decision_engine/multi_agent/configuration*).
5. The selected Decision Engines register to the created topics and send through them acknowledge messages to the supervisor Decision Engine.
6. Once the supervisor Decision Engine receives all expected acknowledgments, it initiates the multi-agent execution by sending a request via each Decision Engine's Access Interface (*POST /api/v1/decision_engine/multi_agent/start*).
7. Each Decision Engine retrieves updated information on available resources from the Telemetry Service within its respective Association. Using speculative and auction-based algorithms, the Decision Engines provide their offered allocations.
8. The supervisor Decision Engine collects responses from the collaborating Decision Engines via the Distributed Data Broker's exchange topics.
9. Based on the received responses, the supervisor Decision Engine determines the distribution of application's microservices across the Associations to meet user requirements while optimizing resource utilization.
10. The supervisor Decision Engine notifies the other Decision Engines of the multi-agent process completion by sending a termination request via their exposed RESTful API (*DELETE /api/v1/decision_engine/multi_agent/configuration/UUID*).
11. The exchange topics created within the Distributed Data Broker are then removed.

12. The Decision Engine forwards placement decisions to the Service Orchestrator, which proceeds with the next phase of the orchestration and deployment process.
13. Finally, the default EMPYREAN Aggregator is informed of these decisions and, if necessary, coordinates with other Aggregators.

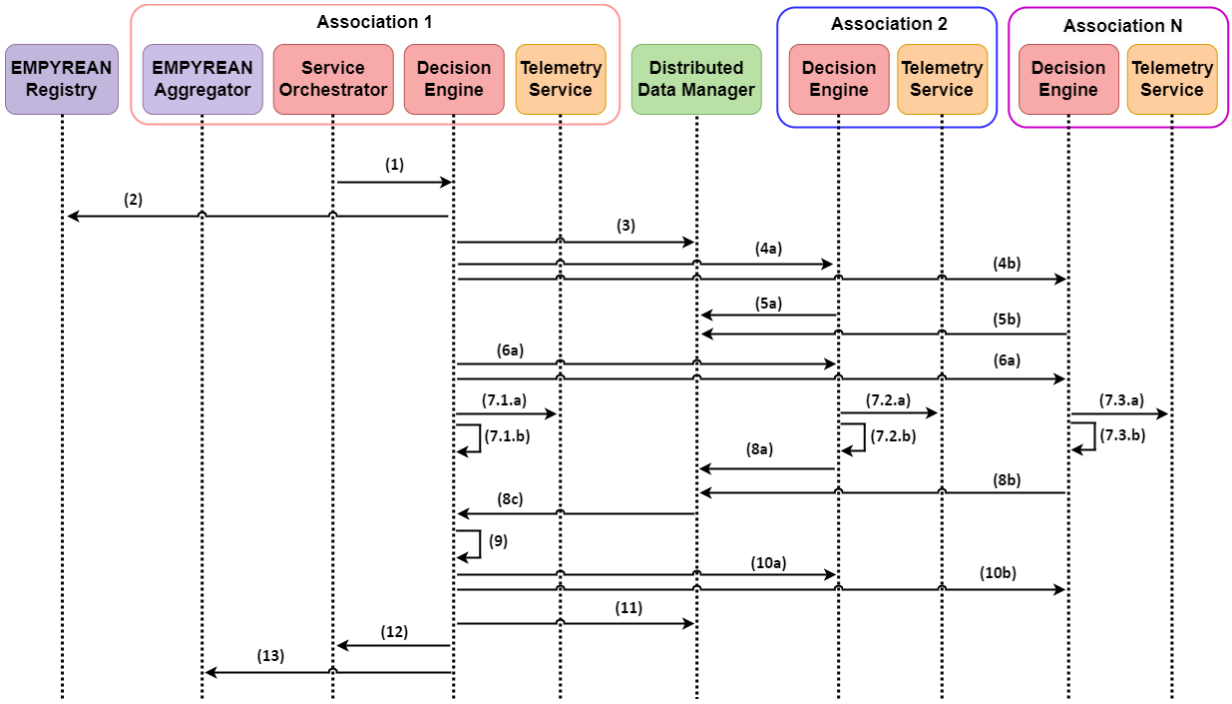


Figure 24: Decision Engine – Multi-agent operation.

6.5 Relation to Use Cases

As an integral part of the EMPYREAN orchestration and deployment mechanisms, the Decision Engine is critical in supporting all project use cases. In most scenarios, its functionalities will be accessed indirectly through the Service Orchestrator, which acts as an intermediary for handling service requests and optimizing resource utilization. Integrating decentralized AI-driven orchestration, multi-agent coordination, real-time telemetry, and energy-aware optimization, the EMPYREAN Decision Engine effectively meets the use cases requirements, contributing directly to their targeted technical KPIs.

To achieve this, the Decision Engine will integrate novel workload assignment and resource allocation algorithms to execute the following key functions: (a) application workload placement across and within Associations, (b) dynamic load balancing of processing and data across Associations, (c) adjusting available computing, networking, and storage resources to meet application-specific demands and QoS requirements, and (d) autonomous Association management. These functions are critical to the control and management plane of the EMPYREAN platform, ensuring efficient resource utilization, enhanced performance, reduced energy consumption, and highly accurate AI-based decision-making.

7 Cyber Threat Intelligence

Cyber Threat Intelligence (CTI) refers to collecting, analysing, and disseminating data regarding potential or existing cyber threats. The EMPYREAN CTI Platform has been designed to streamline and enhance cyber threat intelligence management. It automates routine and time-consuming tasks, allowing security experts to concentrate on critical data and high-priority analysis, while providing real-time, actionable insights through relevant reporting mechanisms. The following section discusses state-of-the-art CTI data analysis and presents the ongoing development of the EMPYREAN CTI Platform.

7.1 State of the art

The analysis of CTI and Indicators of Compromise (IoC) makes clear the use of legitimate services such as CDN, Cloud Services, Instant Messaging, File Sharing Systems for the propagation of malicious files or malicious links related to C2C architecture (Command and Control) or malware infection. Among these services, we find AWS, Dropbox, Google Docs, and Discord. This poses a difficulty in countering the problem with traditional blocking methods. In this scenario, different works have attempted to characterize the main problem, the quality of the Security Information and Event Management (SIEM) dataset, the methodology for analyzing and detecting URLs, malware infrastructure organization, and trends. In this section, we list previous efforts to define methodologies and propose an active solution for all challenges related to this topic using Machine Learning.

Network entities and organizations have implemented countermeasures to prevent attacks by blocking content previously identified as malicious or suspicious by other entities that have suffered such attacks. However, the lack of standardization in how they should report their incidents limits the ability of other entities to take advantage of such previous experiences. To resolve this limitation, different organizations have standardized how to share CTI information in recent years with the Structured Threat Information Expression (STIX) format. The STIX format represents incidents in entity-relationship graphs connecting different significant attack components for a specific threat. Initiatives such as QRadar[5] or OpenTaxi[6] make this type of forensic information public in STIX format. However, only some private initiatives, such as the Cyber Threat Alliance, use this information to improve cybersecurity solutions.

STIX datasets have already been leveraged in different ways. A notable trend among CTI is to group different sources provided in the form of textual reports or lists of indicators of compromise into a Semantic Entity Database. For example, [64] proposes a Unified Cybersecurity Ontology (UCO). Then, several works are based on similar concepts (i.e., ontologies) to retrieve knowledge graphs from external CTI sources (including STIX providers) and apply semantic queries. STIX knowledge graphs are often used as search engines from

[5] https://www.ibm.com/docs/en/qsip/7.5
[6] https://opentaxii.readthedocs.io/en/stable/

which assumptions can be derived that help and enhance the work of a human expert. In [65], an external STIX dataset is used to derive a new database schema and extract well-defined security rules in standardized formats such as YARA and Snort. Therefore, STIX-based graphs are used prominently as databases to perform user-defined queries.

However, these works depend on constructing ontologies from a structured database and integrating them with an external knowledge source. This implies an additional phase of ontology construction and entity retrieval, often obtained through the analysis of plain text sources. This is the case of the work in [66], which proposes using a heterogeneous information network instead of a canonical extraction of Resource Description Framework (RDF) triples to build the base graph. This configuration is then used to perform a downstream task, such as predicting the maliciousness of a domain that interacted with network entities in the graph. Therefore, given the increasing use of the standard and the well-defined entity-relationship model, it is possible to avoid the semantic web architecture and build the graph by applying the standard rules, enriching it, when necessary, with custom external fields, and adding redundant information.

The study in [70] examines the value of commercial threat intelligence, revealing minimal overlap between different providers and open feeds. Paid services often have delayed and limited coverage, raising concerns about their timeliness. Interviews with clients indicate that they prioritize workflow optimization over threat detection, assessing threat intelligence informally rather than using quantitative metrics. [71] investigates the COVID-19 Cyber Threat Coalition (CTC), a voluntary security information sharing community with over 4,000 members. It examines if large-scale collaboration improves coverage and if free threat data enhances defenders' abilities. The CTC largely aggregates existing industry sources but demonstrated unique value by including COVID-19-related domains unknown to current abuse detection systems. The findings offer three lessons for future threat data sharing initiatives.

[82] focuses on classifying attacks in Infrastructure as a Service (IaaS) cloud environment, particularly those involving Virtual Machines (VMs), using Virtual Machine Introspection (VMI) techniques. The classification method considers the source, target, and direction of attacks, helping cloud actors deploy suitable monitoring architectures. The paper includes statistical analysis on reported vulnerabilities and their financial impact on business processes, emphasizing the significance of these attacks in IaaS clouds. The authors of [69] uncover security risks posed by obsolete NS records in active domains, particularly those within the domain's zone. They demonstrate practical exploitation of these records, leading to stealthy domain hijacking. Analysis of high-profile domains, DNS hosting providers, and public resolver operators identifies numerous vulnerable entities, including government bodies, payment services, Amazon Route 53, and CloudFlare. The paper also discusses mitigation techniques for affected parties, offering a comprehensive understanding of this new security risk.

[80] presents a system that learns regular expressions to extract Autonomous System Numbers (ASN) from router interface hostnames, incorporating topological constraints and PeeringDB data. By altering an existing method, the accuracy of ASN extraction improves,

increasing agreement levels with inferred ASNs from 87.4% to 97.1% and reducing errors. This research broadens possibilities for inferring router ownership based on evidential data.

The study in [89] investigates malware's abuse of web applications for attacker-controlled servers. Delays in incident responder and web application provider collaborations facilitate malware proliferation. The authors develop Marsea, an automated malware analysis pipeline, identifying 893 malware instances across 97 families and showing a 226% increase since 2020. They note a 13.7% decrease in malware relying on attacker-controlled servers and successfully collaborate to dismantle 50% of malicious web application content.

[74] provides a longitudinal measurement of the malicious file delivery ecosystem on the web, analyzing network infrastructures and files downloaded over various periods (one day, one month, one year). It identifies two distinct ecosystems: one for delivering potentially unwanted programs (PUP) and another separate network for malware. Despite mostly disjointed ecosystems, there is a crossover between the two. The study reveals biased proportions of PUP to malware, periodic malicious network activity, and offers insights for improving takedown techniques.

[83] presents a detailed analysis of malicious URLs hosted on Twitter, highlighting challenges in blocking resources when legitimate file-sharing platforms are used. It underscores Twitter's poor countermeasures regarding timeliness and coverage. [68] analyzes a dataset of URLs from SIEM Threat Intelligence platforms, proposing grouping them into attack campaigns with shared characteristics. A significant finding is the persistence of numerous malicious URLs remaining active even after being marked as malicious. The study in [81] addresses the proactive identification of malicious URLs, emphasizing ML techniques for efficient in-memory detection. Authors evaluate detection rates and false positives by analyzing a 6,000,000 URL dataset over 48 weeks, providing insights into the evolving malware landscape and internet attack vectors.

[91] assesses VirusTotal file labeling challenges by reviewing 115 academic papers, categorizing common methods used by researchers, and evaluating the dynamics of anti-malware engine labels. Daily snapshots of VirusTotal labels for files from 65 engines show the benefits of threshold-based label aggregation. The study reveals underperforming 'trusted' engines, correlated engine clusters, and false positives and suggests improved data annotation practices.

In analyzing Private Information shared with third-party domains by mobile applications, the study in [86] provides metrics on domain popularity, geographic distribution, categorization, etc. Focusing on Potentially Harmful Applications (PHA) on Android, it offers related works on malware distribution analysis and domain characterization that are useful for future investigations. Building on previous work, [75] examines malware delivery operations' responses to takedown attempts. Findings highlight the prevalence of distributed delivery architectures, the significance of identifying 'super-binaries,' and behaviors post-takedown. The study emphasizes improving security strategies, service provider coordination, and threat monitoring techniques. [79] focuses on the attacked side of the malware delivery ecosystem by setting up isolated virtual machines infected by various droppers. The work examines the

temporal behavior of droppers and downloaded malware, trying to correlate victim characteristics with malicious software choices left by droppers.

Using a large dataset from a mobile security product, [85] applies graph representation learning to predict future installations of Potentially Harmful Applications (PHA) on Android devices. It highlights the extensive collection and transmission of sensitive data to third parties by mobile applications, and reveals regional variations in data collection, raising concerns about regulation and accountability.

[73] provides a comprehensive analysis of malware distribution networks and adversary strategies. Analyzing a dataset of 99,312 binary malware samples from 38,659 distribution sites over 287 days, it reveals current trends in malware distribution, clustering based on file similarity, and distribution site details. Conclusions offer insights into adversary strategies and future directions for combating malware distribution. [78] introduces a downloader graph abstraction to differentiate between benign and malicious downloader trojans. By analyzing 19 million downloader graphs from 5 million hosts, researchers identify indicators of malicious activity. A machine learning system achieves a true positive rate of 96.0%, false positive rate of 1.0%, and detects malware earlier than existing antivirus products.

The study in [87] analyzes 328 million reports for 235 million samples collected over a year, characterizing VirusTotal's (VT) file feed and clustering methods. Feature value grouping (FVG) clustering scales well and produces high-precision clusters, aiding in detecting potentially malicious samples. [77] presents a risk analysis method for malware distribution networks (MDN), examining their structural characteristics and network centrality. By identifying central malware sites and assessing global dynamic risk, the model predicts potential cyberattacks with high accuracy based on initial MDN risk levels and connectivity changes. Addressing malware distribution and installation challenges, [76] presents Nazca, a system detecting infections by analyzing collective network traffic. Nazca focuses on malicious network infrastructures and achieves low false-positive rates, bypassing coverage gaps in reputation databases while being resilient to code obfuscation. [88] focuses on identifying landing pages tied to drive-by download attacks in Malware Distribution Networks (MDN). It proposes a feature selection approach to detect malicious landing page content, expanding understanding of MDNs. The system achieves a confirmation rate of 57% for predicted landing pages, extending MDN footprint by 17%.

Leveraging the dataset from [74], this study in [84] proposes a prediction algorithm based on word embedding techniques for security events. It analyzes CVE numerically, tracking evolution over time, and explores dynamic temporal embedding analysis. This method offers a new way to examine CVEs without feature engineering, potentially applicable to other STIX model nodes. Introducing a Bayesian label propagation model for malware detection in malicious distribution graphs, [67] combines file download relationships and network topology. The evaluation shows efficient and accurate detection without source code inspection, confirming effectiveness on real-world download events. [72] develops a machine learning approach to detect URLs hosted by exploit kits, based on analyzing workflows of 40 kits. WEBWINNOW uses attack-centered and self-defense behavior features for detection, demonstrating high effectiveness with minimal false positives in real-world data.

7.2 Cyber Threat Alliance Platform

We obtain data from the Cyber Threat Alliance (CTA) exchange, where CTA members send threat intelligence to share it and where they download threat intelligence shared by others. All CTA members have access to the data in the CTA exchange, including data sent before they joined the CTA. To maintain access to the exchange, CTA members need to keep a minimum average weekly contribution.

7.2.1 STIX

The data in the CTA exchange is encoded using the Structured Threat Information Expression (STIX) standard, version 2.0, which is machine-readable. Although STIX 2.1 was officially approved in June 2021, it has not yet been adopted by the CTA. There are other competing standards for threat intelligence sharing, such as Malware Information Sharing Platform (MISP) and Incident Object Description Exchange Format (IODEF). The use of STIX in the CTA exchange indicates that STIX has considerable industry support as the leading standard for threat intelligence sharing.

The STIX 2.0 data model revolves around JSON-formatted objects. Each STIX object has a universally unique identifier (UUID), an object type, and a set of properties (name, value) (i.e., fields). The STIX 2.0 standard defines 14 object types. It also allows creating custom object types, but the CTA exchange only supports the 14 standard object types. STIX defines a set of standard object properties, but it also allows defining custom properties that must begin with the prefix *"x_"*. The standard defines common properties that apply to all object types, such as the creator's identity *("created_by_ref"*), creation timestamp (*"created"*), and last modification timestamp *("modified"*). It also defines specific properties for each object type and which common and specific properties of each object are required (i.e., mandatory) or optional. For example, the *"malware"* object must contain a *"name"* property that identifies a malware (e.g., Win.Trojan.Agent-1).

7.2.2 The STIX graph model

STIX uses a graph model with two main types of objects: STIX Domain Objects (SDOs), which correspond to the graph nodes, and STIX Relationship Objects (SROs), which explicitly define the edges. Of the 14 standard object types, 12 are SDOs and 2 are SROs. There are two ways to create edges between objects: (1) embedded relationships and (2) explicit relationships defined by SROs.

Embedded relationships are object properties that contain the UUID of a target object, which may be part of the same package or part of a previously sent package. Embedded relationships implicitly define the source object for an edge, which is the object containing the embedded reference. For example, the *"created_by_ref"* property, required in each object, contains the UUID of an *"identity"* object that captures the entity that created the object. That *"identity"* object may be included in the sent package or may have been sent in a previous package.

The STIX standard defines two SROs to capture explicit relationships: *"relationship"* and *"sighting"*. The "relationship" object contains references to a source SDO and a destination SDO, as well as the type of relationship between both objects. For example, the "indicator indicates malware" relationship captures that the observation of a source "indicator" object indicates a specific type of malware. The other SRO is the "sighting" object, which asserts that an object was observed. It captures which SDO was sighted (e.g., "indicator", "malware", "campaign"), who sighted it (i.e., an "identity" object), and what was actually observed represented as an array of "observed-data" objects.

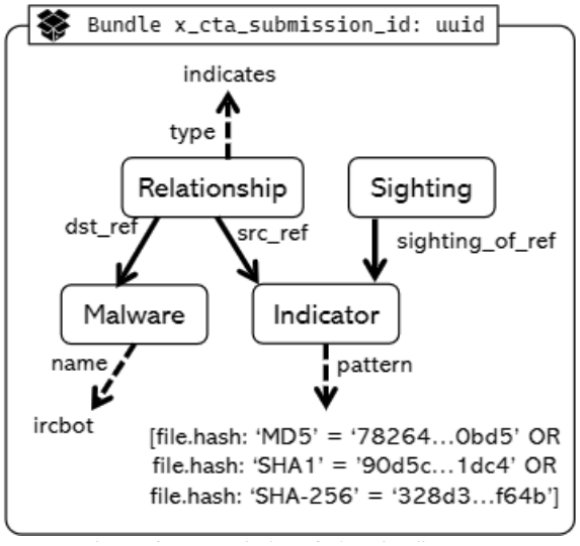


Figure 25: Example of a graph of a bundle in STIX.

Figure 25 shows a graph with four objects: a "sighting" of an "indicator" (a file captured by its MD5, SHA1, and SHA256 hashes) linked to a "malware" family (ircbot) through a "relationship" object (indicates). The edges are annotated with their corresponding property. Solid edges correspond to embedded properties of SDOs (i.e., "sighting_of_ref") or properties of SROs (e.g., "src_ref", "dst_ref"). Dotted edges represent properties that contain object values. We omitted the edges due to the "created_by" property, which would link each object to the "identity" of its creator.

7.2.3 IoCs

In STIX terminology, indicators of compromise (IoC) such as URLs, domains, IP addresses, and file hashes are called "cyber observables". Confusingly, the "indicator" object does not directly contain IoCs. Instead, it contains a detection pattern that may include multiple IoCs. Due to licensing restrictions on popular pattern languages like those used by YARA and Snort, STIX opted to define its own pattern language. The STIX pattern language supports detection rules for both host-based and network-based tools, allowing complex patterns that combine expressions of comparing IoC values with operators (e.g., AND, OR, FOLLOWED-BY) and quantifiers (e.g., REPEATS x TIMES, WITHIN x SECONDS). For example, the pattern "network-traffic:protocols = 'UDP' AND ipv4-addr:value = '112.184.32.238' AND network-traffic:dst_port = '7724'" captures traffic sent to the IP address 112.184.32.238 on destination port 7724/udp.

7.2.4 Data exchange process

The CTA has developed and maintains its own exchange software called "Magellan", which provides a repository of STIX objects and a REST API that allows, among other things, sending STIX objects for sharing, obtaining statistics on sent objects, and querying specific objects. In the past, a Trusted Automated Exchange of Intelligence Information (TAXII) server was also provided, but currently, the data is only accessible through the Magellan platform.

CTA members send batches of STIX objects called "bundles". The STIX standard states that objects in the same bundle do not need to be semantically related, although CTA members may create separate bundles for distinct events.

Each bundle sent is validated, scored, and stored. The validation step requires each bundle sent to contain a "sighting" object, an "indicator" object, and any of the following objects: "attack-pattern", "campaign", "intrusion-set", "malware", "threat-actor", or "tool". Validation also requires "indicator" objects to have the optional "kill_chain_phases" property and all "sighting" objects to have the optional "first_seen" and "last_seen" dates.

The CTA scores each bundle sent, assigning a predetermined number of points to required and important optional properties, while other optional properties do not provide points. The sum of the scores of all bundles sent by a member during a week determines whether the member is contributing the expected minimum volume of threat intelligence.

During the submission process, the CTA adds a series of custom properties to the objects, such as a unique submission identifier for the bundle ("x_cta_submission_id") and the identity of the CTA member submitting the bundle ("x_cta_submitted_by"). The sender may differ from the creator ("created_by_ref") if the CTA member is simply forwarding an object created by someone else. The bundle score ("x_cta_score") is added to the mandatory "sighting" object.

7.3 CTA Data Analysis

7.3.1 Contribution by member

This section compares each member's contribution (anonymized) to the CTA exchange. Each object specifies its creator using the standard property created_by_ref and its sender using the custom property x_cta_submitted_by, introduced in January 2021. Throughout the period, we observed 54 creators and 38 submitters. All 38 submitters are among the 54 creators. We used the CTA API to verify the 16 UUIDs of creators who do not appear as submitters. Of these, 4 are CTA members and the other 12 are unknown entities that create objects forwarded to the CTA exchange by a member. Overall, we observed 42 entities related to the CTA: 41 CTA members and the CTA administrators. Table 6 summarizes the creators, submitters, and members by year.

Table 6: Contribution by year.

Entities	All	2020	2021	2022	2023
Creators	54	31	39	46	38
Submitters	38	-	31	36	35
Members	41	24	27	32	34

To obtain a single submitter per object, from now on, we define the submitter of an object as the UUID in the property x_cta_submitted_by if it exists (i.e., after January 2021), otherwise we use the UUID of the creator in the property created_by_ref if it belongs to the 42 CTA entities, otherwise we set it to null, which only occurs for 0.001% of the objects.

For each member, we calculate their activity by obtaining the first and last day the member submits an object, the lifespan in days obtained by subtracting both dates, the number of days the member contributes objects, the total number of bundles sent, and the total number of objects in those bundles. We also calculate the daily average of bundles and objects by dividing those totals by the lifespan in days.

On average, a CTA member contributes 3,694.6 daily bundles containing 22,186.2 objects. The median number of objects in a bundle is 6.0, a low value indicating that most members use separate bundles for objects that belong to the same incident. However, the average number of objects per bundle is 198.2 because submissions by 5 members contain hundreds or thousands of objects, indicating that their bundles contain unrelated objects from different incidents.

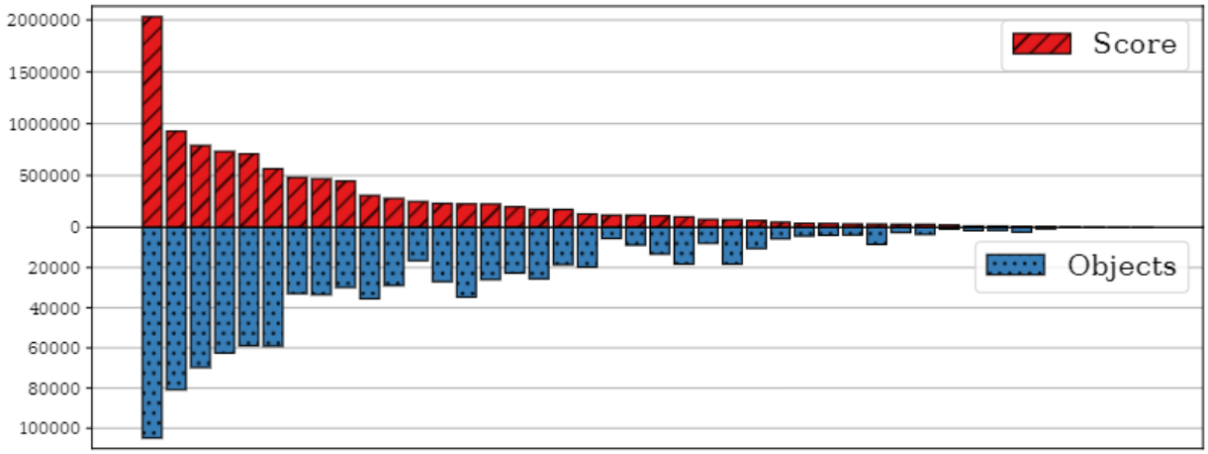


Figure 26: Member contribution: average daily CTA score (above) and average daily contributed objects (below). Members can contribute as much as they want above the minimum required score.

Figure 26 shows, for each member, the average daily CTA score (in red above zero) and the average daily volume of objects (in blue below zero). Daily values are obtained by dividing the total number of objects the member contributes and the sum of the CTA scores in all contributed sighting objects, by the number of days they are a member.

The plot shows significant differences among members. While the top contributor clearly dominates the CTA score, the contribution of objects is more balanced, with 9.6% of objects for the top contributor compared to 8.6% for the second. The four most active members contribute more than a third (34.4%) of the objects and the top 10 members two-thirds (66.3%). Some members have a higher rank for the CTA score compared to their contributed objects, indicating that they may focus on maximizing the score of their submissions.

CTA membership has been increasing over the years, from 24 members in 2020 to 34 in 2023. Only 7 members have left the alliance. Of these, two seem to leave due to being acquired. The other three are the members with the lowest daily average of contributed objects and the lowest fraction of active days, possibly indicating that they could not maintain the minimum required contribution. On average, a member submits objects 86% of the days. Therefore, submissions arrive daily (with some gaps due to downtime or maintenance windows).

The long membership periods and low dropout rate indicate that shared data is considered valuable, although being seen as a member of a select club of leading security providers may also play a role in membership.

On average, a CTA member contributes more than 22K objects daily. Some members contribute well above the minimum required, with 4 members contributing a third (33.4%) of the objects and 10 members contributing two-thirds (66.3%). Most members use small bundles with a median of 5.4 objects that likely capture an event. But 5 members send large bundles of probably unrelated objects.

7.4 EMPYREAN CTI Platform Design

The platform is conceived as an integral tool that supports process automation and report customization, ensuring that security experts can access contextualized and relevant information in real-time. Below, the technical features, operational benefits, and expected impacts of this tool in the field of digital security are detailed.

7.4.1 Periodic data download

The platform for report generation is designed to handle and process large volumes of data efficiently and systematically. Data collection is performed daily, capturing vital information from multiple entry points to ensure all relevant data is considered in cybersecurity analyses. Data is automatically collected each day and sent to a dedicated machine hosting a MongoDB database. This NoSQL database system is chosen for its ability to handle large amounts of unstructured data and its flexibility in managing it, essential characteristics to adapt to the varied forms of data in cybersecurity.

Once stored, the data is accessible through an advanced platform that runs the necessary algorithms for analysis and report generation. This platform is equipped with data analysis tools that enable the execution of complex queries and trend analyses, as well as real-time detection of attack patterns and anomalies. The integration of MongoDB with this platform facilitates a smooth and efficient workflow, where data is processed and visualized in ways that directly support decision-making and incident response.

This infrastructure not only optimizes speed and accuracy in report generation but also enhances the capability of security experts to act based on operational and strategic insights derived from continuous data analysis.

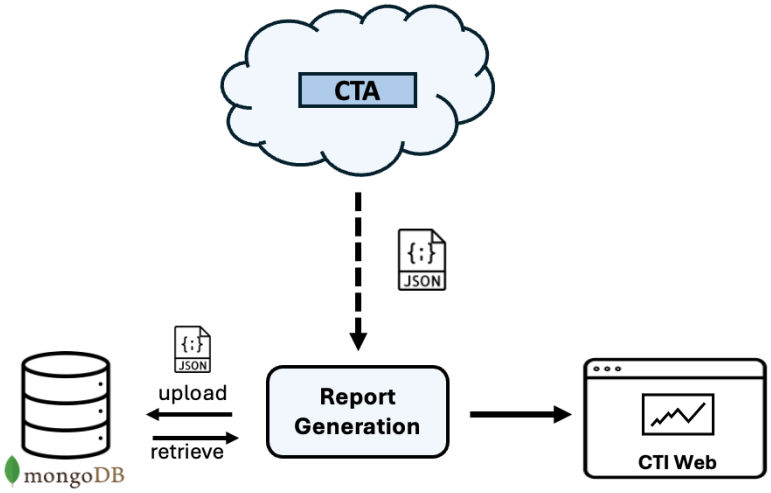


Figure 27: Data analysis platform architecture.

Figure 27 illustrates the operational infrastructure of the cybersecurity report generation platform. Data is collected from the CTA cloud and sent in JSON format to a MongoDB database, where it is stored and managed. Through the ***Autoreport*** module, this data is processed and transformed into reports accessible via the CTA Web platform interface, allowing users to visualize and analyze security information in real-time.

Figure 28 shows the platform's operation mode. In it, we employ a series of automated scripts that play a crucial role in data collection and analysis from the Cyber Threat Alliance (CTA). These scripts are programmed to run daily, ensuring that the latest and most relevant data is systematically downloaded without manual intervention. This automation process is fundamental to maintaining the continuity and currency of the information we handle, which is essential in the dynamic field of cybersecurity.

Each day, the automated scripts activate to connect with the CTA, using secure APIs and data transfer protocols to download the latest updates. These data include detailed threat reports, indicators of compromise (IoCs), and other essential metrics critical for real-time security analysis. Once the data is downloaded, its integrity and format are verified to ensure it is fit for subsequent analysis and storage.

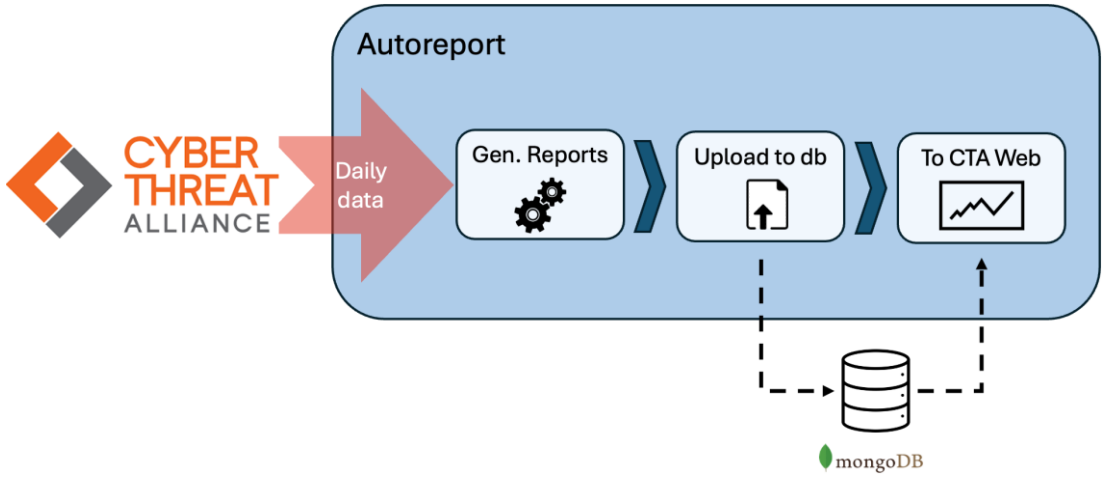


Figure 28: Data analysis platform architecture.

Once collected, the data is immediately subjected to thorough analysis using a set of algorithms designed to detect patterns, trends, and anomalies. This analysis is performed both daily and weekly, allowing our analysts and end-users to gain a comprehensive view of the threat landscape on different time scales. The analysis scripts not only evaluate the frequency and severity of threats but also categorize the data according to various criteria, such as type of threat, origin, and potential impacts.

Once processed, the analysis results are stored in a MongoDB database. This database has been selected for its efficiency in handling large volumes of unstructured data and its capability for high-speed operations. MongoDB facilitates quick and efficient access to data, which is crucial for agile responses to emerging cyber threats. The database structure is optimized for fast queries and data retrieval, enabling efficient management of the vast volume of information we handle.

The data stored in MongoDB directly feeds our web platform, CTA Web, where users can search and access detailed reports. The web interface is designed to be intuitive and easy to use, allowing users to filter and search specifically for the information they need without delays. In addition to searches, users can set up personalized alerts and view data visualizations that help interpret threat trends more effectively.

7.4.2 Data analysis

We have dedicated considerable effort to developing and implementing advanced analysis algorithms that allow us not only to capture and report the most common indicators of compromise (IoCs) but also to discover and understand underlying trends in cybersecurity data. These algorithms are at the heart of our ability to provide contextualized and accurate threat analysis, which is essential for proactive cybersecurity management.

We are currently actively working on perfecting our algorithms. This continuous improvement process ensures that our analysis methods remain at the forefront of threat detection and assessment. Our developers and data analysts collaborate closely to integrate the latest research and data analysis techniques into our processes.

One of the most innovative aspects of our algorithm suite is its ability to identify trends in IoC usage. Beyond simply listing the most frequent IoCs, our algorithms are designed to detect significant changes in the frequency of use of these indicators over time. For example, we can identify an IoC that has been relatively uncommon in the past but has seen an increase in use in the last week. This ability to track the dynamics of IoCs allows users to anticipate and react more effectively to emerging threats.

In addition to IoCs, our algorithms are also equipped to calculate and report the most used vulnerabilities and types of malware. By analyzing large datasets, we can identify which vulnerabilities are being exploited most frequently and which malware are on the rise. This information is not only crucial for immediate incident response but also for strategic long-term defense planning.

Another important facet of our analytical technology is the ability to summarize content. These algorithms are designed to process and condense obtained information, transforming large volumes of data into concise and understandable summaries. This process allows users to quickly gain a clear view of the situation without manually examining vast amounts of data. This capability facilitates the quick identification of key points and significant trends in security data.

7.4.3 Data presentation

This section presents the early mockups of the web interface designed to allow users to visualize the data.

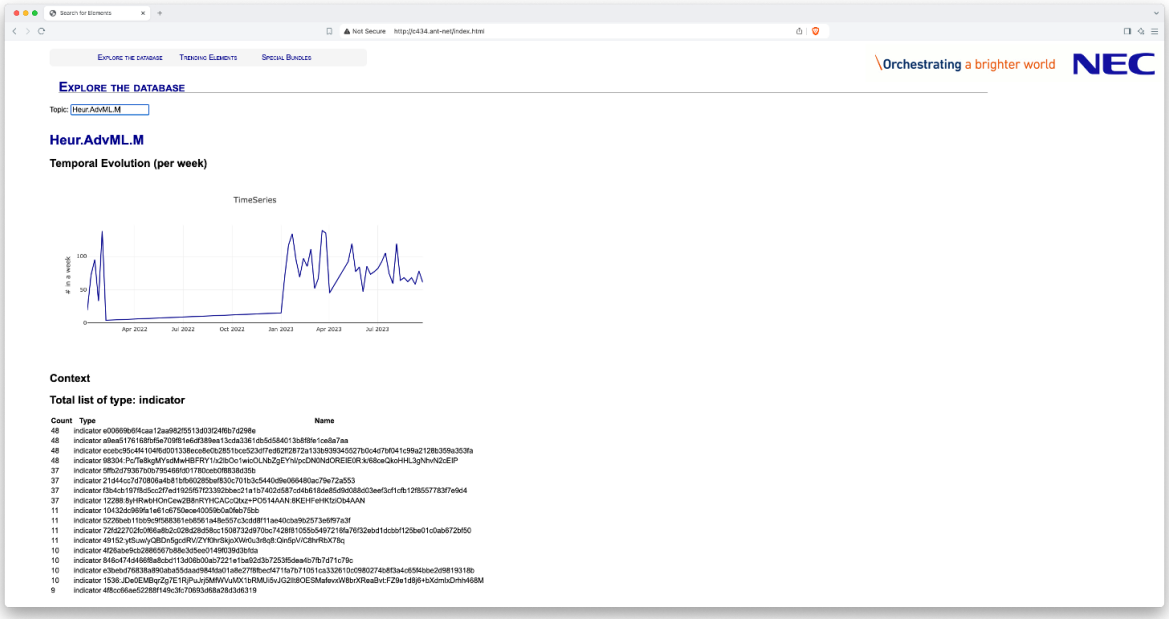


Figure 29: Web mock-up to explore information.

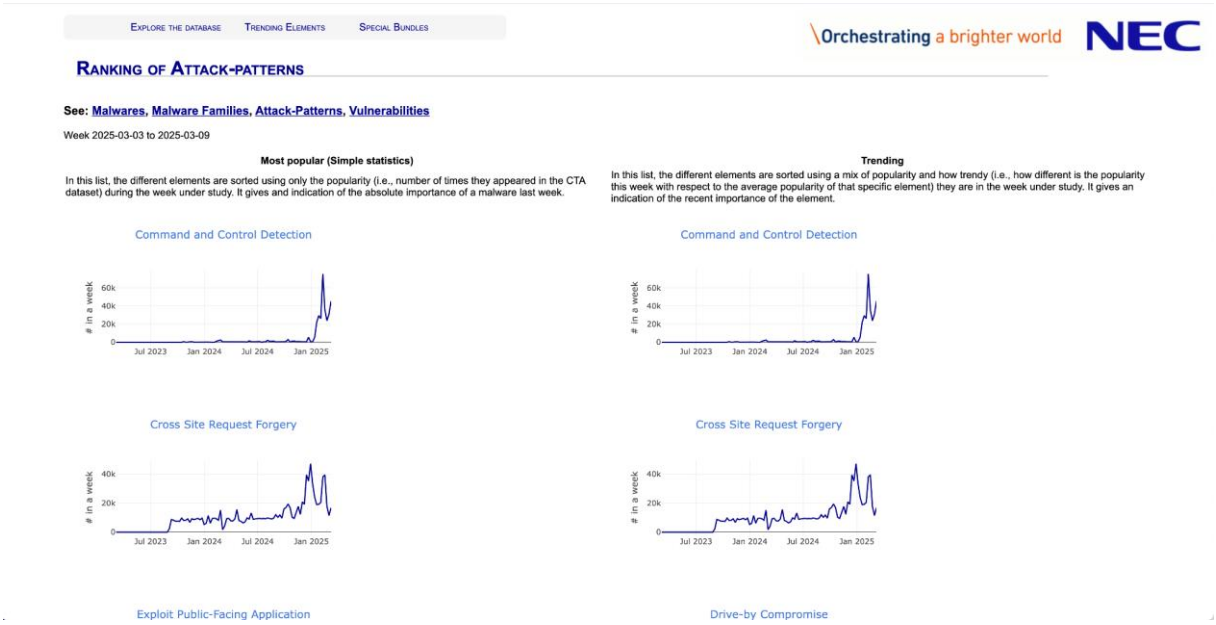


Figure 30: Web mock-up to observe trends in Attack-patterns.

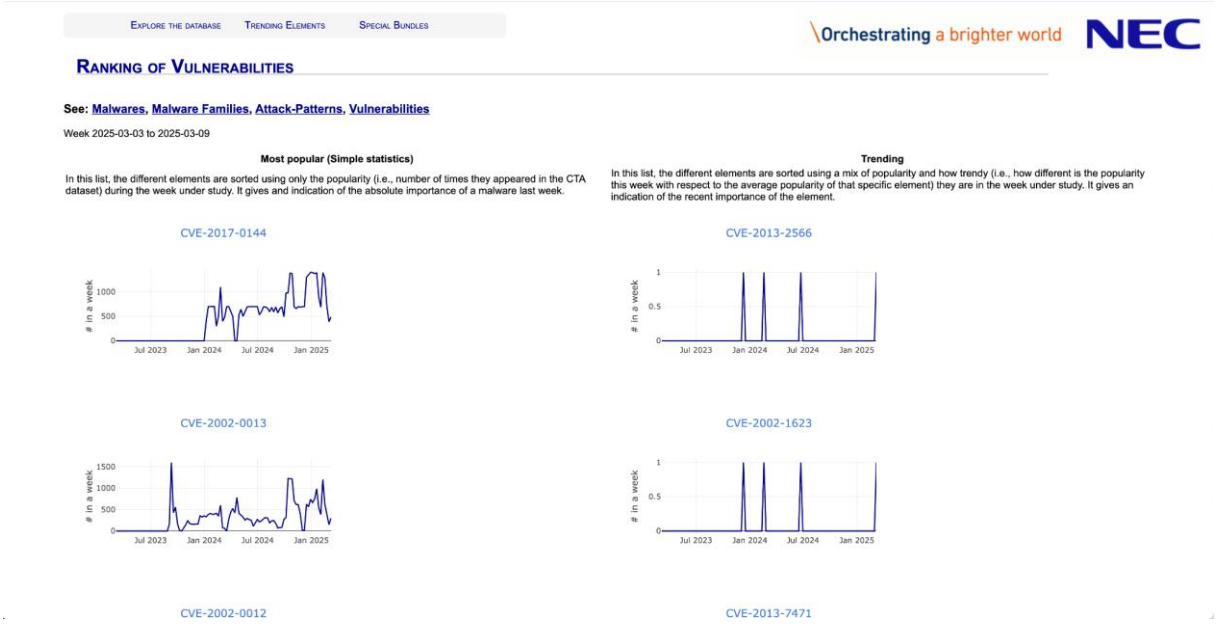


Figure 31: Web mock-up to observe trends in vulnerabilities.

8 Service Orchestrator

8.1 Overview

Managing and deploying services across distributed computing infrastructures is inherently complex due to their heterogeneity. Moreover, the seamless integration of computing and storage resources across the disaggregated IoT-edge-cloud continuum requires automated and intelligent orchestration mechanisms. To address these challenges, the EMPYREAN control plane introduces advanced services for cognitive and efficient orchestration, providing optimized resource management for cloud-native applications.

The Service Orchestrator operates within an EMPYREAN Association, coordinating multiple platform-specific container orchestration systems such as Kubernetes (K8s) and Lightweight Kubernetes (K3s). These systems manage distinct segments of the infrastructure, while the Service Orchestrator ensures abstracted and unified service deployment through hierarchical and distributed orchestration mechanisms at the Association-level.

At its core, the Service Orchestrator leverages the Decision Engine (Section 6) to optimize workload distribution across available platforms within an Association, aligning deployments with application-specific requirements. Once the high-level placement is determined, the Service Orchestrator delegates the final deployment decisions, such as selecting Worker Nodes, to Local Orchestrators on the target platforms. This process follows a declarative approach, where workload requirements are specified to Local Orchestrators rather than issuing direct imperative commands.

In addition, EMPYREAN Controllers, also known as Orchestration Drivers, are deployed on each platform to handle low-level interactions with the platform-level mechanisms and their Local Orchestrators. This layered approach abstracts infrastructure heterogeneity, providing efficient management across distributed environments while ensuring seamless interoperability between edge and cloud platforms.

8.2 Relation to EMPYREAN Objectives and KPIs

The Service Orchestrator and EMPYREAN Controllers are key components of EMPYREAN's cognitive and distributed service orchestration and deployment framework, providing seamless service deployment and efficient operation within EMPYREAN Associations. To address the platform's key objectives and technical KPIs, these services leverage a combination of hierarchical orchestration, cognitive workload placement, and adaptive resource management.

Below is a breakdown of how their design and implementation contribute to each related KPI:

- *T1.1 - Reduce cloud and increase edge utilization via workload balancing optimization*: The Service Orchestrator, through its integration with the Decision Engine, assesses workload placement based on real-time resource availability and application requirements. It ensures optimal distribution of workloads, reducing reliance on centralized cloud resources. The EMPYREAN Controllers enforce these decisions locally, dynamically adjusting deployments to maximize edge utilization while preventing resource saturation.
- *T1.3 - Increase statistical multiplexing gains through associations*: The Associations Deployment model allows multiple EMPYREAN Associations to share resources dynamically, increasing statistical multiplexing across edge and cloud infrastructures.
- *T1.4 - Provide low and predictable latency for hyper-distributed applications*: The EMPYREAN Controllers enable near-real-time application deployment at the edge, ensuring low-latency responses. The Declarative approach in workload placement at the platform-level ensures that service provisioning adapts dynamically to latency constraints without manual intervention.
- *T2.1 - Improve overall performance compared to SotA*: The Service Orchestrator's hierarchical architecture improves scalability and efficiency compared to traditional monolithic orchestrators. Moreover, the distributed orchestration approach increases computational throughput, enabling better performance compared to state-of-the-art orchestration solutions.
- *T2.3 - React fast to rapid changes in computational and data demands so as to maximize the number of demands served*: The EMPYREAN Controllers support event-driven orchestration, allowing rapid redeployment and scaling based on real-time telemetry data. The Datastore's event-based notifications mechanism ensures that changes in computational demand trigger immediate corrective actions.

8.3 Architecture

The Service Orchestrator and EMPYREAN Controller are built upon the Resource Orchestrator service, originally developed by ICCS within the H2020 SERRANO[7] EU project. The Resource Orchestrator is a high-level orchestration framework designed to operate seamlessly across diverse cloud and HPC platforms. In EMPYREAN, its design and implementation are being extended to support the requirements of an Association-based and collaborative IoT-edge-cloud continuum. Additionally, a new EMPYREAN Controller tailored for K3s platforms is introduced, along with a high-level Python API to abstract the deployment and management of cloud-native applications.

[7] https://ict-serrano.eu

The Service Orchestrator (Figure 32) is a cloud-native application implemented in Python, comprising two core services: the *Orchestration API Server* and *Orchestration Manager*. The *EMPYREAN Controllers* (Orchestration Drivers) and the *Datastore* complete the architecture. The Datastore is a critical component for the overall operation and coordination among the Service Orchestrator services and EMPYREAN Controllers.

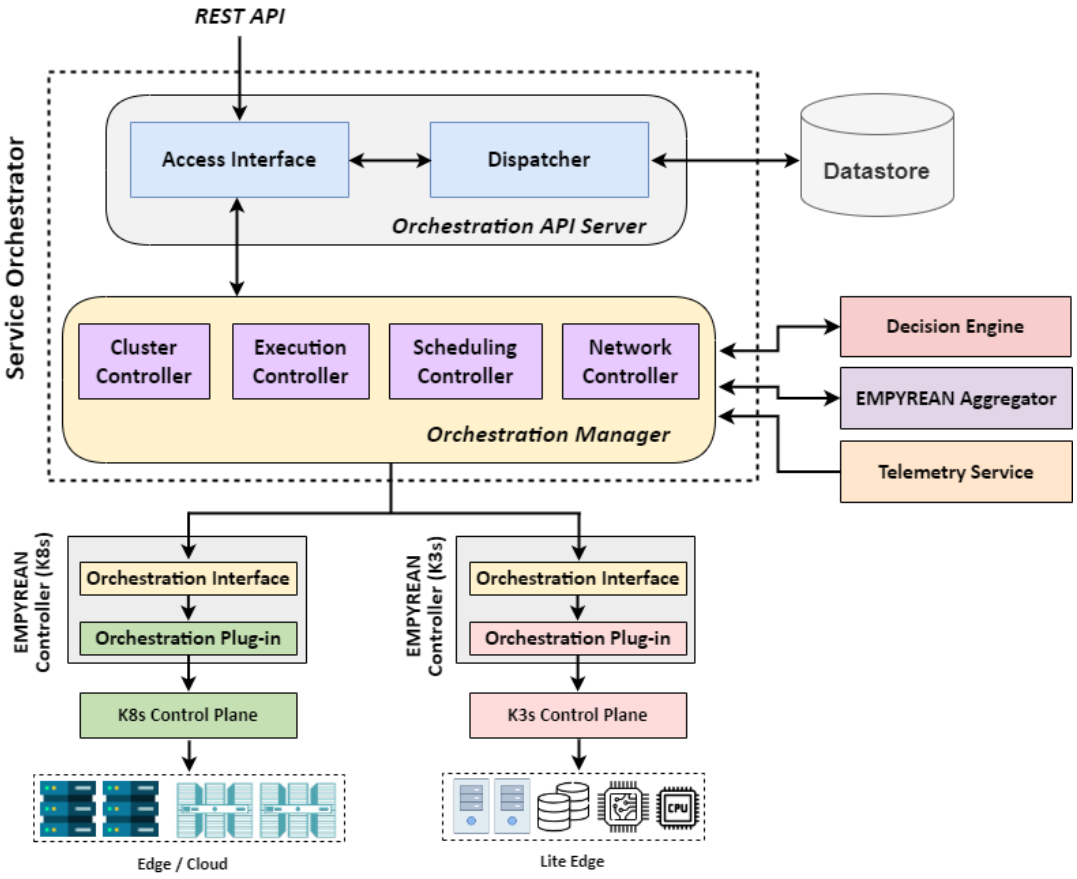


Figure 32: Service Orchestrator and EMPYREAN Controller architecture and its main components.

The Orchestration API Server includes the *Access Interface* component, which implements the necessary mechanisms for bidirectional communication, enabling the exchange of commands, information, and notifications. Before forwarding requests to the *Dispatcher,* the Access Interface validates them. The exposed RESTful API (Section 8.4) supports deployment and management of cloud-native applications and facilitates autonomous interactions between the Orchestration Manager, EMPYREAN Controllers, and the Datastore. The *Dispatcher* exclusively manages interactions with the Datastore, enabling the creation, management, and query of Orchestration API objects along with the subscription of external services to watch specific topics in the Datastore.

The Orchestration Manager is responsible for coordinating the workload placement and initiating application deployment and service provisioning. It operates based on Orchestration API objects (Section 8.4) created by the Orchestration API Server. The Orchestration Manager incorporates four specialized controllers that monitor specific topics and Orchestration API

objects within the Datastore. These controllers execute the required operations to serve requests, interact with other EMPYREAN control plane services, such as the Decision Engine, the EMPYREAN Aggregator, and the Telemetry Service, and communicate with the EMPYREAN Controllers managing the underlying platforms.

The *Cluster Controller* attaches K8s and K3s clusters to the Service Orchestrator and oversees their operational state. The *Scheduling Controller* interacts with the Decision Engine to retrieve the instructions for the cognitive application deployment. The interaction is based on the REST API exposed by the Decision Engine Controller (Section 6.4). The *Network Controller* enables dynamic network service provisioning by interacting with dedicated network management services. Finally, the *Execution Controller* prepares the required application deployment instructions (declarative approach) with the assistance of the Scheduling and Network controllers, coordinates the required data movement by interacting with the storage services within the EMPYREAN platform, and finally triggers the actual deployment by interacting with the EMPYREAN Controllers at the selected edge and cloud platforms.

The Datastore is based on etcd[8], an open-source distributed key-value store, and maintains configuration and state data for the Orchestrator API objects. A key feature of etcd is its "watch" function, accessed through the Watch API, which provides an event-driven interface for asynchronously monitoring changes to stored keys. This functionality is utilized in the design of the Service Orchestrator to facilitate communication between the Orchestration API Server, Orchestration Manager, and EMPYREAN Controllers (Orchestration Drivers). By leveraging this event-driven approach, the Service Orchestrator can continuously track both the actual and desired state of deployed applications and across the unified infrastructure.

There are two types of EMPYREAN Controllers, both following a common design (Figure 32). The *Orchestration Interface* component provides an infrastructure-agnostic layer between the Service Orchestrator (i.e., Orchestration Manager) and local orchestration mechanisms. It enables a generic representation of application description, deployment preferences and constraints. The *Orchestration Plug-in* translates the infrastructure-aware deployment objectives from the Service Orchestrator into platform-specific requests for the local orchestration mechanisms. This component varies for each EMPYREAN Controller type and is designed to interact with the APIs exposed by each local orchestration platform.

8.4 Implementation

The Service Orchestrator and EMPYREAN Controller components are implemented in Python and packaged as container images, ensuring an efficient and modular development and deployment workflow. The Orchestration API Server, Orchestration Manager, and the available EMPYREAN Controllers are each deployed as separate container images to enhance scalability and maintainability.

[8] https://etcd.io

The REST API exposed by the Orchestration API Server is organized into two main categories. The first set of methods (Figure 33) supports the deployment and management of cloud-native applications and facilitates the provision of services. The second set (Figure 34) abstracts the interaction of the Orchestration Manager and EMPYREAN Controller services with the Datastore by enabling them to create, update, and query relevant information along with their subscription for watching specific topics in the Datastore.

In the Orchestration API server, the functionality of the Access Interface and Dispatcher components has been enhanced to support provisioning and management of cloud-native applications within the EMPYREAN Association-based continuum. Specifically, the northbound interface of the Service Orchestrator has been extended with new methods that enable the definition and management of deployment requests across multiple Associations (methods at endpoint */api/v1/service_orchestrator/associations/deployments*). Additionally, the REST API responsible for inter-component communication has been expanded to support newly introduced Orchestration API Objects (i.e., Associations, and Application) for EMPYREAN to ensure seamless integration.

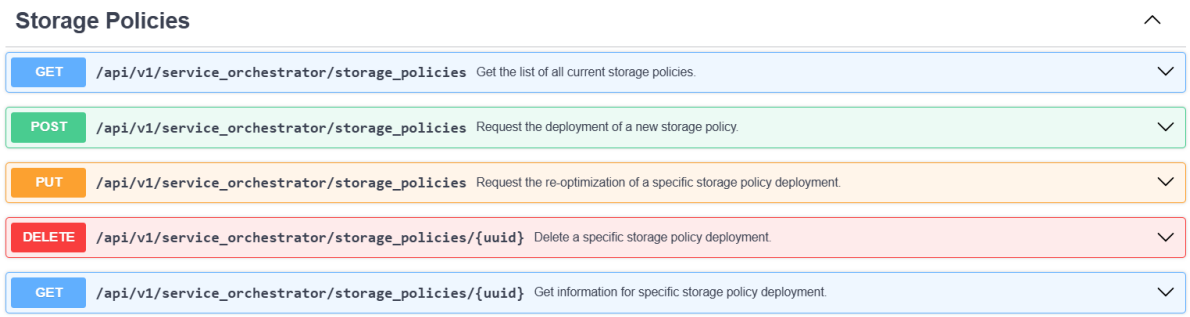


Figure 33: Service Orchestrator REST API.

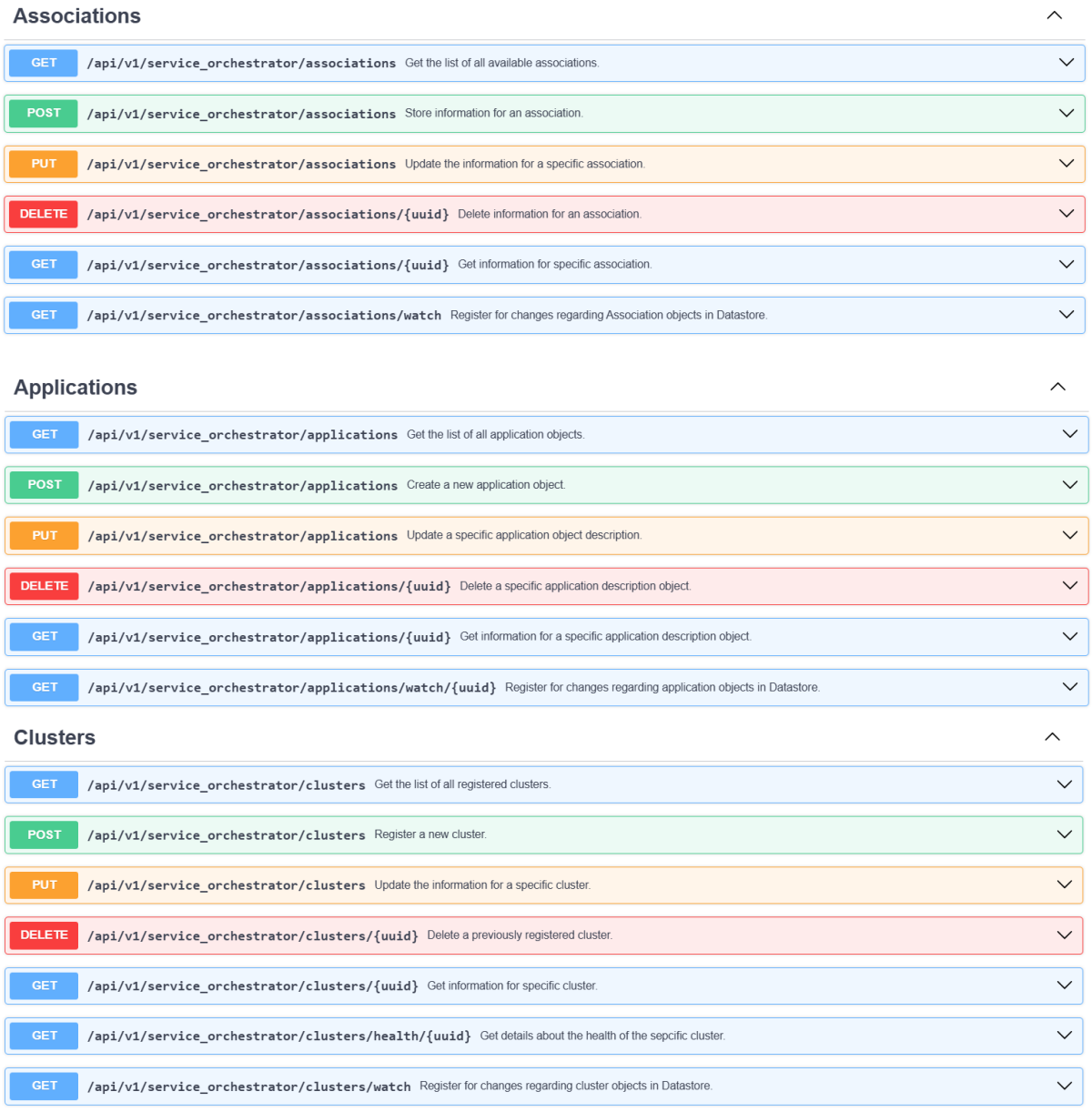


Figure 34: Service Orchestrator REST API – Methods related to inter-component communication.

Service requests are expressed as Orchestration API objects (Figure 35) that serve as the primary communication mechanism between different system components. These objects encapsulate all necessary information for serving, managing, and monitoring service requests. Specific pairs of Service Orchestrator services are responsible for creating, updating, deleting, and watching these Orchestration API objects to facilitate the interaction between the orchestrator services. This distributed responsibility model optimizes request handling and enables seamless system operation.

The main Orchestration API objects are the following:

- ***Associations Deployment***: It represents the high-level description required for deploying a cloud-native application across multiple EMPYREAN Associations. This object includes the application description along with the user-defined deployment objectives. It is created and deleted by the Orchestration API server, while it is watched and used by Orchestration Manager.
- ***Deployment:*** It defines the deployment description of cloud-native application within a specific EMPYREAN Association. It is created and deleted by the Orchestration API server, while it is watched and used by Orchestration Manager. These entities are immutable during the final orchestration and deployment phases (OF4.1.2 and OF4.1.3) since application's microservice distribution into available clusters and infrastructure-specific instructions are expressed through the Assignment and Bundle objects.

- ***Storage Policy***: It specifies the high-level description for creating and managing a storage policy. It is created and deleted by the Orchestration API server and watched and used by the Scheduling and Execution Controllers within the Orchestration Manager that execute all the required operations.
- ***Application***: It corresponds to the complete application description, including its microservices, their characteristics, dependencies, and interconnections. This object is created in case of multi-Association deployment and is used by the Orchestration Manager.
- ***Association***: It provides an overview of the active Associations that are managed by the specific Service Orchestrator. These objects are created and updated based on the information from the EMPYREAN Aggregator and used by the Orchestration Manager.
- ***Cluster:*** It provides an overview of the available platforms (edge, cloud). These objects are created and updated based on the information from the EMPYREAN Controllers while watched and used by the Orchestration Manager.
- ***Assignment:*** It is an internal object that captures the assignment of application microservices to a specific edge/cloud cluster. They are created, updated, and deleted by the Orchestration Manager according to the decisions from the Decision Engine while they are watched and used by the EMPYREAN Controllers (Orchestration Drivers).
- ***Bundle:*** It includes the microservices description along with parameters and platform-specific deployment objectives based on Decision Engine suggestions that will guide the low-level orchestration mechanisms at the selected platforms (declarative approach). These entities are created, updated, and deleted by the Orchestration Manager and they are watched and used by the EMPYREAN Controllers.

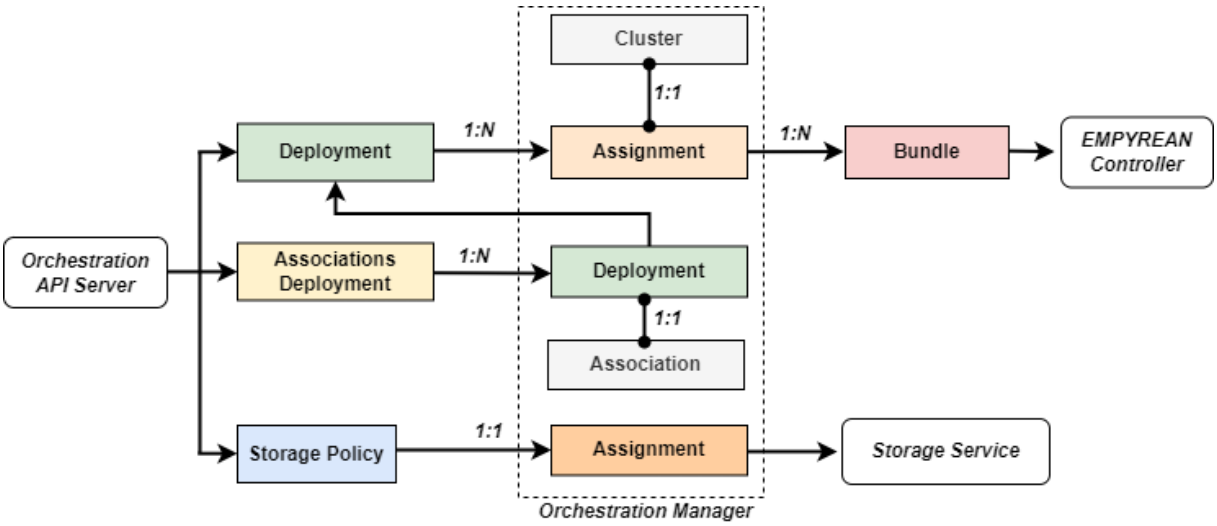


Figure 35: Orchestration API objects and their relationship.

Figure 35 presents the relationships and interactions between the Orchestration API objects, illustrating how these components are interconnected within the system. Complementing this visual description, Table 7 lists the relevant topics for each Orchestration API object stored and managed within the Datastore. These topics ensure proper coordination, management, and functionality of the orchestration mechanisms within an Association and across Associations, contributing to the overall efficiency and reliability of the EMPYREAN platform.

Table 7: Datastore topics (keys) for the main Orchestration API objects.

API Object	Topic
Associations Deployment	/service/orchestrator/associations_deployments/deployment/DEPLOYMENT_UUID
Deployment	/service/orchestrator/deployments/deployment/DEPLOYMENT_UUID
Storage Policy	/service/orchestrator/storage_policies/policy/POLICY_UUID
Application	/service/orchestrator/applications/application/APPLICATION_UUID
Association	/service/orchestrator/associations/association/ASSOCIATION_UUID
Cluster	/service/orchestrator/clusters/cluster/CLUSTER_UUID
Assignment	/service/orchestrator/assignments/CLUSTER_UUID/assignment/ASSIGNMENT_UUID
Bundle	/service/orchestrator/bundles/bundle/BUNDLE_UUID

The EMPYREAN Controllers are available in two distinct types (i.e., K8s and K3s) both implemented in Python as plug-ins. They share a common Orchestration Interface implementation, while their Orchestration Plug-in component differs based on the controller type. This distinction is necessary because the Orchestration Plug-in translates infrastructure-specific deployment objectives from the Service Orchestrator into platform-specific instructions for the local orchestration mechanisms. Each plug-in interacts with the respective local orchestration platform through its exposed API.

For example, the K8s Orchestration Plug-in communicates with the Kubernetes API Server (kube-apiserver) to manage platform-level services, requesting the deployment and management of container-based workloads based on the objectives defined by the Service Orchestrator and Decision Engine. Similarly, the K3s Orchestration Plug-in interacts with the K3s API Server, managing platform resources such as Deployments, Pods, ConfigMaps, and Services.

The workflow in Figure 36 outlines the operation of an EMPYREAN Controller. During its initialization phase, it registers with the Orchestration API Server, subscribing to watch for any changes related to assignments in its dedicated topic in the Datastore. It also reports a summary of available resources on its managed platform, allowing the Orchestration API Server to update the corresponding Datastore entries. The controller periodically sends heartbeat messages to the Orchestration API Server to confirm availability. These steps are common across all EMPYREAN Controller types and are handled by the Orchestration Interface using a set of methods that all Orchestration Plug-ins must implement.

When a change occurs in its subscribed topic, the EMPYREAN Controller is notified. Based on the event type, it triggers the appropriate actions to serve the request from the Orchestration Manager. To this end, it formats the appropriate instructions to the underlying control plane and forwards them using the corresponding exposed API. The EMPYREAN Controller continuously tracks the deployed requests and notifies the Orchestration Manager whenever the actual state differs from the desired state.

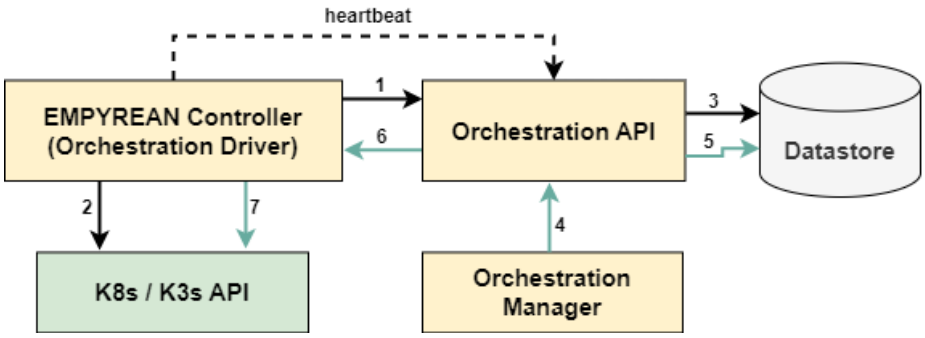


Figure 36: EMPYREAN Controller operation workflow.

8.5 Relation to Use Cases

The Service Orchestrator and EMPYREAN Controllers enable a cognitive, distributed, and adaptive orchestration framework that optimizes resource utilization, enhances performance, and ensures resilience in heterogeneous IoT-edge-cloud environments. Across all use cases, the Service Orchestrator and EMPYREAN Controllers provide efficient, collaborative, and adaptive application deployment that meets the specific needs of each domain, from industrial manufacturing to agricultural sensing and smart factory security.

UC1 - Anomaly Detection in Robotic Machining Cells: The Service Orchestrator ensures that computational tasks related to anomaly detection models are placed close to robotic cells at the edge, minimizing latency and enabling real-time decision-making. The EMPYREAN Controllers dynamically allocate resources to support adaptive AI inference pipelines, balancing execution between on-site edge devices and cloud resources when higher computational power is required. The orchestration framework supports event-driven data processing, enabling fast response times to detected anomalies, ensuring system reliability in high-precision manufacturing environments.

UC2 - Proximal Sensing in Agriculture Fields: The Service Orchestrator facilitates the deployment of proximal sensing applications across dispersed edge nodes within agricultural fields, ensuring efficient data collection from various sensors. The EMPYREAN Controllers manage workload distribution based on real-time environmental data, optimizing sensor-driven computations at the edge while leveraging cloud resources for historical trend analysis and AI model training. The orchestration mechanisms enable fault tolerance, ensuring continuous operation even in environments with intermittent network connectivity by dynamically reconfiguring workloads across available compute nodes.

UC4 - Security in Smart Factories (S. Korea International Collaboration): The Service Orchestrator plays a key role in deploying cybersecurity services across distributed factory infrastructures, ensuring real-time threat detection and response at both edge and cloud layers. The EMPYREAN Controllers support automated scaling of security monitoring applications based on anomalous behavior patterns, adapting to real-time variations in network activity. The platform's ability to provide low-latency orchestration ensures that security policies and protective measures are enforced dynamically across global industrial deployments, maintaining data integrity and operational continuity.

9 Telemetry Service

9.1 Overview

The EMPYREAN Telemetry Service is a core enabler of intelligent and adaptive management within the hyper-distributed and federated environment of the EMPYREAN platform. It is designed to provide seamless, real-time monitoring and observability across the full IoT-edge-cloud continuum, ensuring that critical infrastructure components, devices, and workloads are continuously supervised and optimized. The telemetry infrastructure is not conceived as a standalone module but as an integrated, dynamic ecosystem that interacts with orchestration, analytics, decision-making, and security components, providing them with high-quality, actionable data to drive autonomous and informed operations across Associations.

The EMPYREAN Telemetry Service delivers comprehensive and adaptive monitoring across the IoT-edge-cloud continuum. It supports continuous discovery of heterogeneous resources, such as IoT devices, robotic systems, edge nodes, and cloud services, enabling dynamic adaptation to infrastructure changes. The service collects real-time performance metrics (CPU, memory, network, and energy usage), providing an up-to-date view of system health. Distributed Telemetry Engines handle pre-processing and filtering to manage high data volumes, ensuring only relevant information is propagated. Data is stored in the Persistent Monitoring Data Storage (PMDS) for historical analysis and informed decision-making. The service also adapts its configurations in response to lifecycle events like deployments, migrations, and failures through integration with orchestration components. By extending established observability tools like Prometheus, Grafana, and InfluxDB to support EMPYREAN's federated architecture, it ensures seamless interoperability in distributed environments. Standardized interfaces expose telemetry streams to components such as the Decision Engine, Analytics Engine, and CTI Engine, enabling advanced analytics, security, and autonomous operations.

The EMPYREAN Telemetry Service goes beyond traditional monitoring solutions by offering a federated observability framework for the IoT-edge-cloud continuum. It maintains global visibility over highly heterogeneous and mobile resources, supporting dynamic and distributed infrastructures. Native energy consumption monitoring enables energy-aware optimization strategies that support sustainability objectives. The service autonomously reconfigures its monitoring processes in real-time, adapting to events like migrations, deployments, and failures without manual intervention. High-fidelity telemetry streams power advanced multi-agent algorithms in the Decision Engine, which coordinate to optimize performance, resilience, and resource usage across Associations. With its scalable, lightweight, and distributed architecture, the Telemetry Service addresses the complexity of hyper-distributed environments beyond the capabilities of existing solutions.

9.2 Relation to EMPYREAN Objectives and KPIs

The EMPYREAN Telemetry Service is fundamental in supporting several of the project's key objectives and technical KPIs. Specifically, the Telemetry Service directly contributes to the achievement of the following KPIs:

- *T1.1*: Supplying comprehensive operational metrics, the Telemetry Service enables workload balancing strategies that reduce reliance on the core cloud and increase edge utilization.
- *T1.2*: Through its continuous observability mechanisms, the Telemetry Service enhances system reliability by detecting failures and performance degradations at the edge.
- *T1.4*: Real-time monitoring and Decision Engine integration enable quick adjustments to maintain low and stable latency in hyper-distributed applications.
- *T2.2*: Collecting detailed energy consumption metrics across all platform resources supports energy optimization, contributing to reduced consumption across Associations.
- *T2.3*: Real-time telemetry streams allow the system to promptly detect and react to sudden changes in workload and resource demands, maximizing the number of successfully served requests.
- *T2.4*: By delivering accurate and high-fidelity telemetry data, the service improves the effectiveness of AI-driven decision-making algorithms, directly boosting their accuracy.
- *T2.5*: The robust monitoring capabilities facilitate the early detection of anomalies and noisy conditions, increasing the overall robustness of algorithmic operations.

Overall, the Telemetry Service is a key enabler of EMPYREAN's mission to deliver an intelligent, adaptive, and resilient continuum platform, ensuring optimal performance, resource efficiency, and reliability across federated environments.

9.3 Architecture

The EMPYREAN Telemetry Service is designed as a distributed architecture composed of three main components that provide comprehensive monitoring, analysis, and storage of telemetry data across the IoT-edge-cloud continuum.

- *Telemetry Engines*: These components manage and orchestrate the Telemetry Service as wells as process telemetry data from different infrastructure segments. Operating in a distributed manner, they ensure a unified and consistent view of system health, pre-process raw telemetry data, and provide relevant insights to other platform components such as the Decision Engine and the Analytics Engine.

- *Monitoring Probes*: Lightweight components deployed across the platform to continuously collect real-time performance metrics, logs, and events from specific resources, including hardware, applications, and services.
- *Persistent Monitoring Data Storage (PMDS)*: Serves as a centralized repository for storing telemetry data over time. It enables historical analysis, trend detection, and long-term resource optimization, supporting the platform's decision-making processes with accurate and timestamped data that can also be used for advanced analytics and reporting.

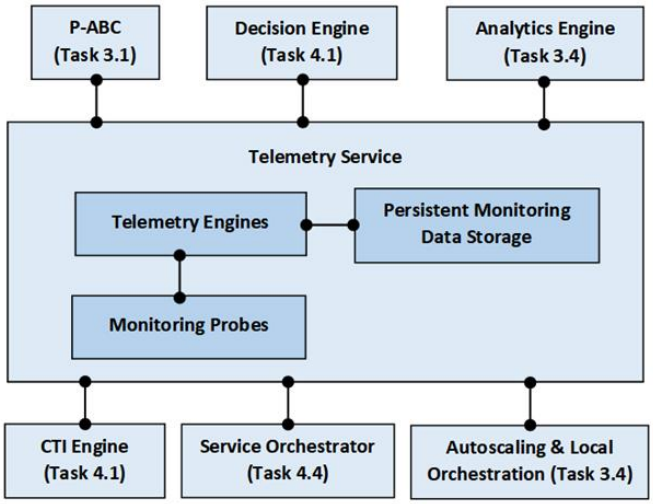


Figure 37: EMPYREAN Telemetry Service components and dependencies.

9.4 Implementation

The EMPYREAN Telemetry Service follows a modular and scalable approach, relying on widely adopted open-source technologies to ensure reliable telemetry collection, processing, and visualization across the continuum. Although the design is still in its early stages, the following components and flow are envisioned:

- *Telemetry Collection*: Metrics will be gathered from multiple sources, including both Kubernetes-based infrastructures and distributed IoT devices. For the cloud-native stack, components such as kube-state-metrics[9] (which collects the state of Kubernetes resources) and Node Exporter[10] (which exposes hardware/system metrics) will be used. For IoT environments, lightweight telemetry clients or gateway nodes will collect metrics and forward them to the monitoring infrastructure using messaging protocols such as MQTT and AMQP. These protocols ensure low-overhead, reliable communication from resource-constrained or intermittently connected devices.

[9] https://github.com/kubernetes/kube-state-metrics
[10] https://prometheus.io/docs/guides/node-exporter/

Specialized adapters or receivers within the OpenTelemetry (OTEL) Collector[11] will subscribe to these data streams and ingest telemetry in a structured format, allowing seamless integration with the rest of the monitoring pipeline.

- *Data Processing and Storage*: The OTEL Collector will apply necessary transformations and filtering to incoming telemetry, ensuring only relevant and clean data is forwarded. This process includes metrics from cloud-native workloads, infrastructure nodes, and IoT devices. The processed data will then be sent to Prometheus[12], the core time-series database, responsible for storing, indexing, and exposing telemetry data.
- *Data Consumption*: Once stored in Prometheus, telemetry data becomes accessible to various components and users across the platform. A REST API exposes metric endpoints, enabling EMPYREAN modules—such as the Decision Engine, Analytics Engine, and Orchestration Engine—to retrieve both real-time and historical data for intelligent decision-making, anomaly detection, and workload optimization. Additionally, AlertManager continuously evaluates telemetry against predefined rules and thresholds, triggering alerts in response to abnormal behaviors, performance degradation, or failures. This approach enables proactive incident response and system self-healing mechanisms. Moreover, for human operators and developers, Grafana[13] will offer rich, dynamic dashboards to visualize telemetry data. These dashboards support custom queries, temporal comparisons, and multi-source overlays, providing deep insights into distributed components' health, performance, and trends across the IoT-edge-cloud landscape.

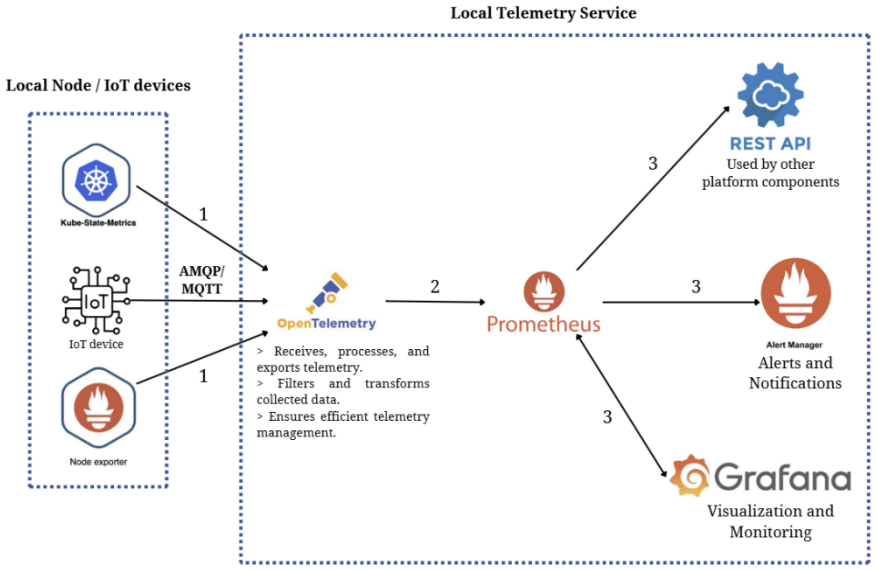


Figure 38: EMPYREAN Telemetry Service implementation.

[11] https://opentelemetry.io/docs/collector/
[12] https://prometheus.io
[13] https://grafana.com

This telemetry pipeline ensures continuous monitoring and analysis of system performance, enabling other EMPYREAN components—such as orchestration, resource management, and security modules—to leverage this information for optimizing operations and responding to potential incidents. Figure 38 summarizes the implementation process described above, illustrating the flow of telemetry data from collection to consumption across the platform.

9.5 Public APIs

The Telemetry Service exposes a public API—*agent API*—that allows external systems to dynamically manage telemetry pipelines within the monitoring infrastructure. Built with FastAPI, this API enables the creation, modification, and deletion of pipelines without manual intervention, making it easy to adapt observability workflows in Kubernetes environments.

Key features include:

- ***Dynamic pipeline management***: Supports registering and updating pipelines by adjusting the configuration of components like the OpenTelemetry Collector, adding new receivers, processors, and exporters as needed.
- ***Lifecycle operations***: Provides endpoints to safely create, modify, and remove pipelines in a controlled way.
- ***Integration with Kubernetes resources***: Works within specific namespaces and manages resources like ConfigMaps to apply changes, whether running inside or outside the cluster.
- ***Support for observability tasks***: Includes operations to inspect component status, list active pods, search by labels, and trigger configuration reloads.

The API is designed to be modular and extensible, with future improvements planned such as automated pipeline adaptation and integration with advanced orchestration systems.

The infrastructure also exposes the *Prometheus public API*, allowing external systems and users to query telemetry data in real-time. This interface supports a wide range of use cases, including resource utilization analysis, anomaly detection, and the creation of custom dashboards. Through the API—using PromQL (Prometheus Query Language)—clients can retrieve metrics such as CPU usage, memory consumption, network traffic, and other key indicators at both system and application levels. These queries can be filtered by time ranges, labels, or resource types, supporting both automated and manual data consumption. The API is also compatible with tools like Grafana, enabling dynamic and interactive data visualization. This makes the Prometheus API a core enabler of observability within EMPYREAN, offering standardized and secure access to performance data across the platform.

Moreover, the EMPYREAN Telemetry Service incorporates the Persistent Monitoring Data Storage (PMDS) component, which enables the long-term retention of collected, timestamped telemetry data. PMDS provides a central repository, retaining the state of heterogeneous resources and deployed applications. This historical telemetry data is critical for feeding the various AI/ML-driven decision-making and analytics mechanisms within the EMPYREAN platform.

The PMDS is implemented using InfluxDB[14], an open-source, high-performance time-series database. To ensure interoperability and facilitate integration with other platform components and external services, PMDS exposes a RESTful API. This API abstracts direct database interactions and offers advanced query capabilities, including time-range filtering, conditional retrieval based on resource or application attributes, and aggregation operations. Through this interface, both internal modules and authorized users can efficiently access and analyze historical telemetry datasets, supporting informed, data-driven operations and enabling continuous system adaptation within the edge-cloud continuum.

9.6 Integration with EMPYREAN Platform Services

The EMPYREAN Telemetry Service is designed to seamlessly integrate with the platform's core services (Figure 39), ensuring efficient data exchange and adaptive observability across the entire system. A key integration point is the Decision Engine, which relies on continuous, high-fidelity telemetry streams to execute complex multi-agent decision algorithms. The Telemetry Service exposes real-time and historical metrics covering resource utilization, performance trends, and energy consumption through standardized public APIs. These APIs allow the Decision Engine to retrieve the necessary data to support workload placement, scaling decisions, and failure recovery, ensuring that management strategies are informed by up-to-date insights from both local and federated environments.

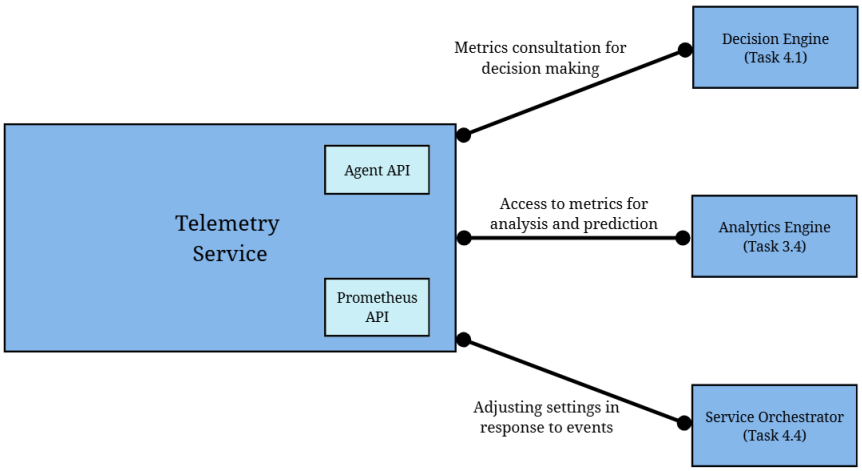


Figure 39: EMPYREAN Telemetry Service integration.

In addition, the Analytics Engine interacts with the Telemetry Service to feed advanced monitoring data into machine learning pipelines and anomaly detection frameworks. By accessing aggregated and pre-processed telemetry through public RESTful APIs, the Analytics Engine can perform deep analysis on system behaviour, identify performance degradations, and trigger predictive actions. This collaboration enhances the platform's ability to adapt

[14] https://github.com/influxdata/influxdb

proactively to evolving workloads and infrastructure conditions, leveraging telemetry as a foundational element for intelligent operations across Associations.

Moreover, the Telemetry Service maintains tight coordination with the EMPYREAN Aggregator and Service Orchestrator, ensuring that monitoring configurations dynamically adapt to lifecycle events such as workload migrations, deployments, and scaling actions. By consuming event notifications from the orchestration mechanisms and exposing APIs to update monitoring policies, the Telemetry Service guarantees that observability remains consistent as the system evolves. This bidirectional interaction not only ensures robust monitoring under dynamic conditions but also provides valuable feedback to the orchestrator, supporting informed resource allocation and scheduling decisions.

9.7 Relation to Use Cases

Next, we provide an overview of how the Telemetry Service integrates with and supports the specific use cases within the EMPYREAN platform. Each use case presents unique challenges that benefit from continuous monitoring, real-time data collection, and advanced analytics provided by the Telemetry Service. By detailing the interactions and contributions of the Telemetry Service in each case, we highlight its role in enhancing system performance, ensuring efficiency, and enabling intelligent decision-making across diverse application scenarios.

Anomaly Detection in Robotic Machining Cells (UC1)

The Telemetry Service plays a central role in the Anomaly Detection in Robotic Machining Cells (UC1) use case by providing real-time monitoring of robotic systems in machining environments. It collects and processes critical data from sensors embedded in the robotic cells, such as tool performance, precision levels, and environmental conditions. The Monitoring Probes deployed within the system continuously gather data, which Telemetry Engines then filter and pre-process. This data is stored in the Persistent Monitoring Data Storage (PMDS) for long-term analysis and historical reference.

The Telemetry Service enables advanced analytics and anomaly detection through integration with the Analytics Engine and Ryax Workflow Engine. When abnormal behaviors are detected—such as tool wear or deviations in machining precision—alerts can trigger automated corrective actions. The service is designed to support both real-time data collection at the deep edge layer and more complex processing at the far edge, ensuring seamless communication and data flow between these layers. The distributed architecture, powered by EMPYREAN's multi-clustering capabilities, guarantees that resources are optimized, and workloads are efficiently managed across edge devices. The telemetry data also supports performance monitoring and helps improve the system's efficiency by informing decision-making and ensuring operational reliability. Moreover, integrating the Telemetry Service with EMPYREAN's security framework ensures data protection throughout the system, allowing for secure and privacy-conscious monitoring in critical manufacturing environments.

Proximal Sensing in Agriculture Fields (UC2)

In Proximal Sensing in Agriculture Fields, the Telemetry Service plays a critical role in monitoring the real-time performance of UAVs and robots involved in soil analysis. By tracking operational parameters such as flight speed, altitude, battery status, and CPU/memory usage, the Telemetry Service ensures that these devices operate efficiently throughout the entire process, minimizing the risk of operational failures or inefficiencies. This is particularly important in dynamic agricultural environments where precision and continuous monitoring are vital.

The Telemetry Service works hand-in-hand with Dataflow Programming to manage the complex data flows between edge devices (UAVs and robots), edge servers, and cloud infrastructure. This guarantees that raw data collected at the edge is efficiently processed with minimal latency and transferred for further analysis or storage. Furthermore, it integrates seamlessly with Edge Storage, ensuring that large volumes of sensor data are initially processed locally, reducing the strain on cloud resources. The Decentralized Data Manager enhances the data exchange, triggering workflows in the Workflow Manager for automated processing and analysis. This combination of components helps optimize soil health assessments, ensuring that actionable insights are available to farmers in real-time, enabling informed, data-driven decisions in agricultural management.

Security in Smart Factories - S. Korea International Collaboration (UC4)

In Security in Smart Factories, the Telemetry Service plays a key role in enhancing the factory's security measures by continuously monitoring various system parameters, such as network traffic, device status, and sensor data. This telemetry data feeds into the Cyber Threat Intelligence (CTI) component, enabling the identification of potential security threats based on abnormal behavior detected through Federated Learning (PPFL).

The Telemetry Service helps aggregate critical information from edge devices and sends it to the central security modules for further analysis. It ensures that the security applications deployed within the smart factory's private 5G network are efficiently orchestrated and scaled, enabling real-time responses to threats. Additionally, this service integrates with MISP (Malware Information Sharing Platform) to share threat intelligence across nodes, fostering a collaborative security environment and enhancing overall system resilience.

10 EMPYREAN Aggregator

10.1 Overview

The EMPYREAN Aggregator serves as a foundational element of the EMPYREAN ecosystem, enabling intelligent orchestration, security, and automation for distributed applications. It establishes the management fabric of the EMPYREAN continuum, enabling seamless coordination across the IoT-edge-cloud infrastructure. By integrating multiple self-managed and interacting Aggregators, the system fosters an autonomous, collaborative, and composable environment where resources and services are dynamically managed and optimized.

The Aggregator employs a hierarchical two-level structure to oversee resource allocation, workload execution, and interconnection across its Associations. Aggregators communicate among themselves but also with edge computing infrastructures and multi-cloud providers, ensuring a cohesive and adaptive management approach. This design enhances operational resilience, allowing Associations to function independently even when connectivity to remote cloud resources is limited or unavailable.

10.2 Relation to EMPYREAN Objectives and KPIs

The EMPYREAN Aggregator is designed to enable collaborative autonomy in the IoT-edge-cloud continuum. Its implementation aligns with the specified project objectives and technical KPIs as follows:

- *T1.1 - Reduce cloud and increase edge utilization via workload balancing optimization*: The EMPYREAN Aggregator prioritizes edge-first computing by leveraging AI-driven workload balancing algorithms. By dynamically offloading tasks to nearby edge nodes, the system minimizes dependency on cloud resources, reducing latency, and data transmission costs.
- *T1.2 - Increase reliability in the edge*: To enhance edge reliability, the Aggregator employs distributed fault-tolerance mechanisms and proactive failure detection via its Analytics and Telemetry Engine. If an edge node experiences performance degradation or failure, it coordinates the workloads' redistribution to alternative nodes within the Association, ensuring service continuity.
- *T1.3 - Increase statistical multiplexing gains through associations*: The Aggregator enables dynamic resource sharing across multiple Associations, allowing workloads to be flexibly assigned based on available capacity, workload type, and priority levels. This multi-agent cooperation enhances statistical multiplexing gains, as computing resources are efficiently pooled and allocated across distributed infrastructures, preventing resource underutilization.

- *T1.4 - Provide low and predictable latency for hyper-distributed applications*: By deploying latency-sensitive applications closer to data sources, the Aggregator ensures that data processing and execution occur within the edge environment whenever feasible.
- *T2.2 - Reduce energy consumption on Associations compared to standard execution*: The EMPYREAN Aggregator facilitates adaptive workload placement that prioritizes execution on energy-efficient edge nodes along with workload consolidation to minimize underutilized resources, reducing overall energy demand.

10.3 Architecture

The Aggregator is designed to consolidate multiple EMPYREAN services and components, delivering essential intelligence and orchestration logic for managing an Association. It operates as an abstraction point between the components of the Service and Multi-Cluster Orchestration layers, ensuring composability and interoperability across the continuum. The Aggregator integrates various functionalities essential for deploying applications, ensuring workload security, managing distributed data storage, and facilitating decentralized interconnection.

The architecture of the EMPYREAN Aggregator (Figure 40) is modular and platform-agnostic, enabling seamless integration with heterogeneous computing environments across the continuum. An Aggregator orchestrates its Associations that include separate or shared computational and storage resources and communications with others to enable the collaborative management of the IoT-edge-cloud continuum.

The EMPYREAN Aggregator is composed of multiple key components, each responsible for a critical function in managing distributed resources and ensuring service reliability. These components include:

- *Service Orchestrator*: Responsible for orchestrating workloads across diverse computing environments, including cloud, edge, and fog layers. It ensures optimized placement of services based on resource availability and policy constraints (Section 8).
- *Decision Engine*: Employs AI-driven mechanisms to enable intelligent decision-making for dynamic resource allocation, fault tolerance, and adaptive workload management (Section 6).
- *Ryax Runner*: Functions as the execution engine for user workflows within an Association. It bridges the Workflow Manager with the resources of individual platforms, enabling seamless execution of actions and workflows. Deliverable D4.1 (M15) provides more details.
- *Edge Storage Gateway*: Provides a distributed and hybrid storage architecture, supporting encrypted and secure data management across edge and cloud layers. It integrates with existing storage systems to facilitate seamless data accessibility. This component is described in deliverable D3.1 (M15).

- *Distributed Data Manager*: Manages decentralized interconnection by ensuring efficient and secure data exchange between distributed components. It incorporates advanced data routing and streaming mechanisms to enhance low-latency performance. Deliverable D4.1 (M15) describes this component.
- *Privacy and Security Manager*: Ensures trust and identity management within the EMPYREAN ecosystem by employing blockchain-based decentralized authentication, secure enclaves, and privacy-preserving cryptographic techniques. This component is described in deliverable D3.1 (M15).
- *Telemetry Engine:* Monitors heterogeneous resources and applications across the distributed infrastructure. It provides real-time observability and generates performance insights to ensure system resilience (Section 9).
- *Analytics Engine*: Implements service assurance mechanisms by continuously analyzing telemetry data, detecting anomalies, and recommending proactive optimizations to maintain system efficiency and reliability. Deliverable D3.2 (M15) details this component.

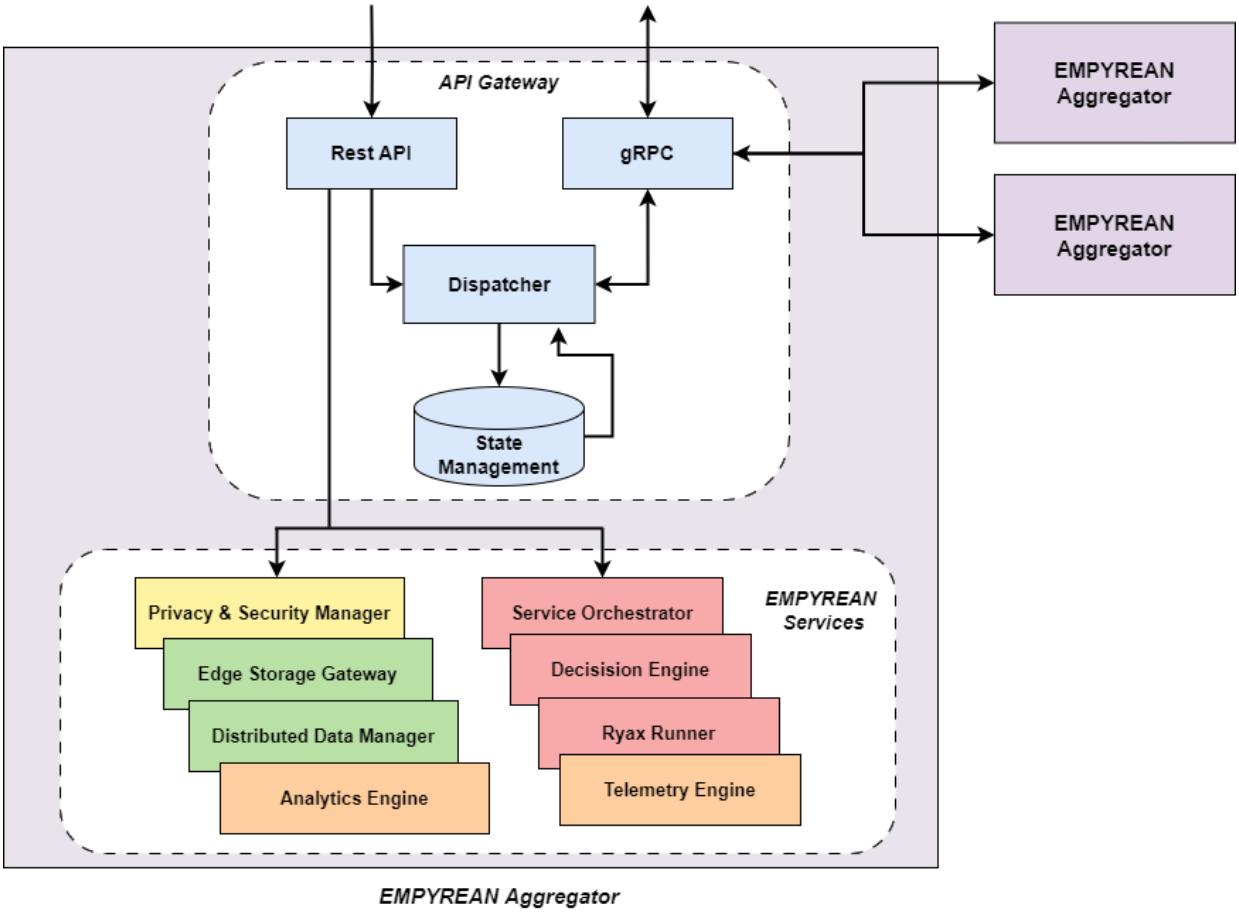


Figure 40: EMPYREAN Aggregator architecture.

The successful integration of EMPYREAN Aggregators and their associated services relies on leveraging the well-defined interfaces within the EMPYREAN platform to ensure seamless communication. To achieve this, the design incorporates the *API Gateway*, which plays a crucial role in enabling secure, scalable, and efficient intra- and inter-Association communication. By providing standardized data models and utilizing the exposed APIs of other services, the API Gateway facilitates service discovery, routing, and management while enforcing stringent policies on access control and authentication.

Furthermore, the API Gateway introduces several key features to enhance system interoperability and performance: (i) API composition, which streamlines interactions by aggregating multiple API calls into a single, optimized request flow, (ii) protocol abstraction, supporting a variety of communication protocols such as RESTful APIs, gRPC, and AMQP (for asynchronous messaging), and (iii) logging and monitoring, achieved through integration with the Telemetry Engine, providing real-time insights into API usage, request latencies, and overall system performance.

10.4 Implementation

The EMPYREAN Aggregator is built using a microservice-based approach, allowing each component to operate independently while being loosely coupled to ensure flexibility and scalability. The Aggregator is implemented as a cloud-native application in Python, where individual components interact through their well-defined APIs. The core services of the Aggregator are containerized and orchestrated via Kubernetes, allowing for easy deployment, dynamic scaling, and fault tolerance.

The API Gateway provides two NBIs, namely a *REST interface* exposed to core EMPYREAN orchestration and management services, such as EMPYREAN Registry and Workflow Manager, that enable them to seamlessly interact with the Associations and resources managed by an Aggregator, and a *gRPC interface* exposed to the rest of the EMPYREAN Aggregators. Both interfaces rely on the *Dispatcher, who* handles the incoming requests and interacts with the *State Management* component. The State Management enables a stateful implementation of the API Gateway to support the operational requirements within the EMPYREAN platform. To enhance performance and responsiveness, the State Management component integrates Redis[15] as a high-speed caching layer and lightweight storage solution for stateful data.

Figure 41 summarizes the available methods in the REST interfaces of the API Gateway component.

[15] https://redis.io

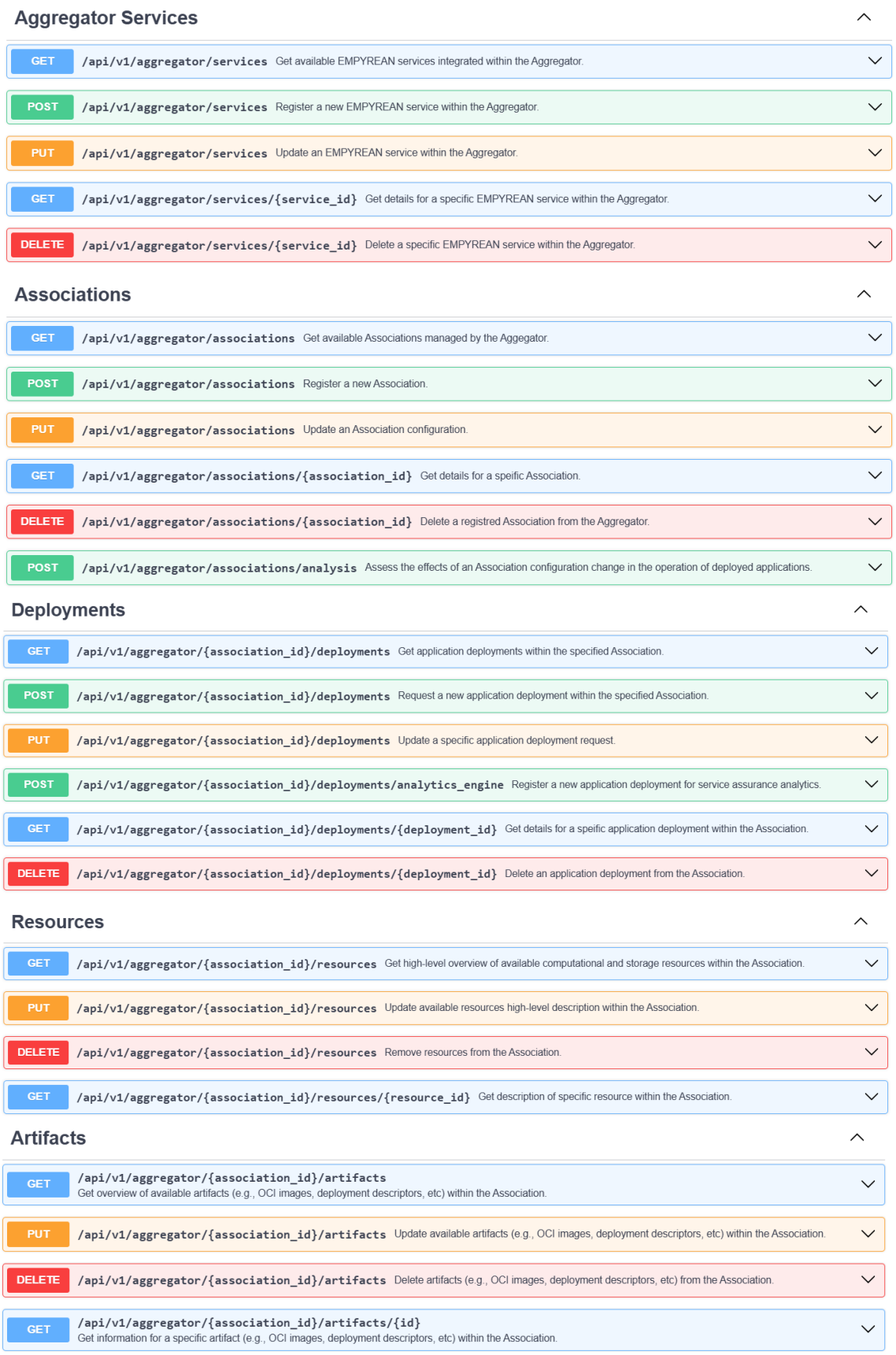


Figure 41: EMPYREAN Aggregator – API Gateway RESTful API.

The API Gateway component also incorporates publish-subscribe mechanisms over gRPC to exchange information, commands, and events among the EMPYREAN Aggregators. This interface will provide methods to enable collaboration among Aggregators to get/set/delete information from/into their databases, as well as to receive asynchronous events reporting on changes within an Association. The RPC methods currently exposed by the initial version of this interface are presented in Table 8. As the implementation progresses towards the initial release of the EMPYREAN platform, this interface will be extended appropriately.

Table 8: EMPYREAN Aggregator – RPC methods.

Method Name	Description
AssociationUpdates	Bidirectional streaming method for real-time updates for available Associations managed from an Aggregator.
ArtifcactsUpdates	Bidirectional streaming method for real-time updates for available artifacts within an Aggregator.
ResourcesUpdates	Bidirectional streaming method for real-time updates for available resources within an Association.
CreateAssociationDeployment	Assign an application deployment to a specific Aggregator.
AssociationDeploymentUpdates	Bidirectional streaming method for real-time updates regarding the progress of an assigned application deployment request.
MigrateDeployment	Request an application deployment migration to some other Aggregator.
MigrateData	Request data migration to an Association to some other Aggregator.
MigrateDeploymentEvents	Bidirectional streaming method for real-time updates regarding the progress of an application deployment migration request.
MigrateDataEvents	Bidirectional streaming method for real-time updates regarding the progress of data migration request.

10.5 Relation to Use Cases

The EMPYREAN Aggregator serves as the intelligent orchestration and resource management backbone for the project's use cases (UCs), ensuring efficient workload distribution, low-latency execution, and secure data handling across diverse IoT-edge-cloud environments. By leveraging its integration with other key components of the EMPYREAN architecture, the Aggregator abstracts the Association-based continuum to the end users, automates infrastructure-specific operations, and optimizes performance and resilience for each UC.

For anomaly detection in robotic machining (UC1), real-time data collection, processing, and analytics are crucial. The Aggregator facilitates low-latency execution of AI models at the edge, ensuring that machine learning-based anomaly detection runs locally for faster responses. In agricultural field monitoring, large volumes of sensor data must be processed efficiently to enable precision farming (UC2). The Aggregator ensures seamless integration of proximal sensing devices, processing data locally via edge computing nodes to reduce cloud dependence and optimize real-time analytics. For smart factory security (UC4), the EMPYREAN Aggregator enhances cybersecurity and access control by integrating decentralized trust management mechanisms. This ensures resilient and secure factory operations, even in highly interconnected industrial environments.

11 EMPYREAN Registry

11.1 Overview

The Association-based continuum represents a dynamic framework where entities—ranging from research institutions and enterprises to edge-cloud platforms—interact seamlessly to enable intelligent, decentralized, and scalable service deployment. The EMPYREAN Registry is essential to support this complex and interconnected environment and address key challenges in the EMPYREAN platform's coordination, management, and governance.

The EMPYREAN Registry serves as a unified entry point for both core platform services and third-party entities, enabling the discovery, cataloguing, and advertising of Associations and services across the Association-based continuum. The Registry facilitates the registration and management of IoT devices, edge, and cloud resources within Associations. Moreover, it keeps track of the available Associations and services, the mapping of infrastructure resources to Associations, and the relationships between users and Associations.

11.2 Relation to EMPYREAN Objectives and KPIs

The initial design and implementation of the EMPYREAN Registry contribute significantly to realizing the EMPYREAN vision of an autonomous, distributed, and collaborative IoT-edge-cloud continuum. As a core component of the AI-driven control and management plane, it supports key EMPYREAN objectives and enables the fulfillment of the following technical KPIs:

- *T1.1 - Reduce cloud and increase edge utilization via workload balancing optimization*: The Registry facilitates intelligent workload placement decisions by supporting the multi-agent operation of multiple Decision Engines across the Associations.
- *T1.2 - Increase reliability in the edge*: Enabling decentralized services and maintaining resource metadata supports adaptive service discovery and redundancy mechanisms at the edge.
- *T1.3 - Increase statistical multiplexing gains through associations*: It supports efficient grouping and reuse of distributed resources, enabling higher utilization and reduced idle capacity across edge and cloud environments.

11.3 Architecture

The EMPYREAN Registry is designed as a scalable, cloud-native system that integrates various functionalities essential for ensuring seamless discovery, cataloguing, and advertisement of Associations and services across the EMPYREAN platform. It is dynamically updated with new information as infrastructure resources are registered or new applications and services are published. The Registry follows a modular architecture (Figure 42) with seven core services.

The *API Gateway* is the central access point for all interactions with the EMPYREAN Registry, ensuring secure, scalable, and controlled access to its functionalities. It handles external and internal service requests, manages authentication and authorization, and facilitates event-driven communication between EMPYREAN services and the core Registry components. The API Gateway's architecture aligns with the respective component in the EMPYREAN Aggregator (Section 10.3), ensuring consistency and interoperability across the platform. The API Gateway exposes both REST and gRPC interfaces, enabling flexible communication models suited for different integration needs within the EMPYREAN ecosystem.

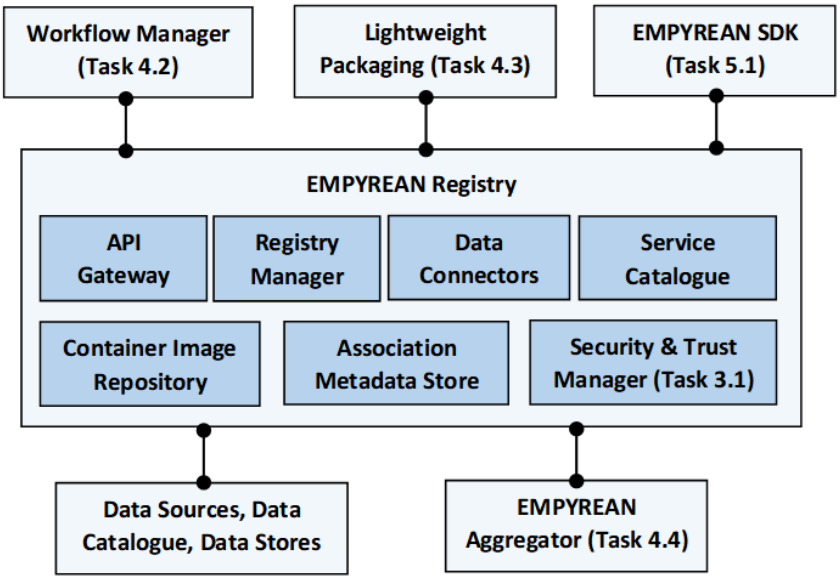


Figure 42: EMPYREAN Registry architecture.

The *Registry Manager* is the core orchestration component of the EMPYREAN Registry, responsible for overseeing its operation and managing its interactions with other services to ensure seamless integration with the broader EMPYREAN platform. It provides lifecycle management, handling the registration, modification, and removal of Associations and services. It also cooperates with the Privacy and Security Service to enable predefined governance policies for access control, data sharing, and compliance across Associations. Moreover, the Registry Manager ensures real-time consistency of Registry data across distributed Associations using event-driven updates.

The *Container Image Repository* stores and manages OCI-compatible images of hyper-distributed applications. These images are built and packaged using EMPYREAN's dedicated mechanisms, ensuring compatibility and efficient deployment across the Association-based continuum. This service distributes container images that support all EMPYREAN container runtime types, providing a single source of truth for containerized applications within the continuum. By integrating with EMPYREAN's development mechanisms, as detailed in Deliverable D4.1 (M15), the Container Image Repository enables fast and secure deployments across heterogeneous execution environments.

The *Service Catalogue* provides a centralized repository for managing application blueprints, deployment configurations, and metadata related to services and datasets. By integrating with the Container Image Repository, the Service Catalogue streamlines application development and deployment across the continuum. It provides a comprehensive service management that stores metadata about software packages, container images, service descriptors, and hyper-distributed application models. It also allows EMPYREAN services and users to search, filter, and retrieve available services in a structured way. The Service Catalogue enables automated service composition and deployment based on predefined metadata attributes and provides inputs for EMPYREAN's orchestration and deployment mechanisms, improving service interoperability and adaptability.

The *Association Metadata Store* contains metadata about available Associations, providing a high-level description of participating resources, their ownership, and sharing policies, along with descriptions of available EMPYREAN services and deployed applications. This component aggregates metadata and capabilities from individual Associations and platforms, making them accessible across multiple Associations in accordance with the selected privacy settings. The orchestration and deployment mechanisms leverage this data to manage Associations and deploy applications within and across them.

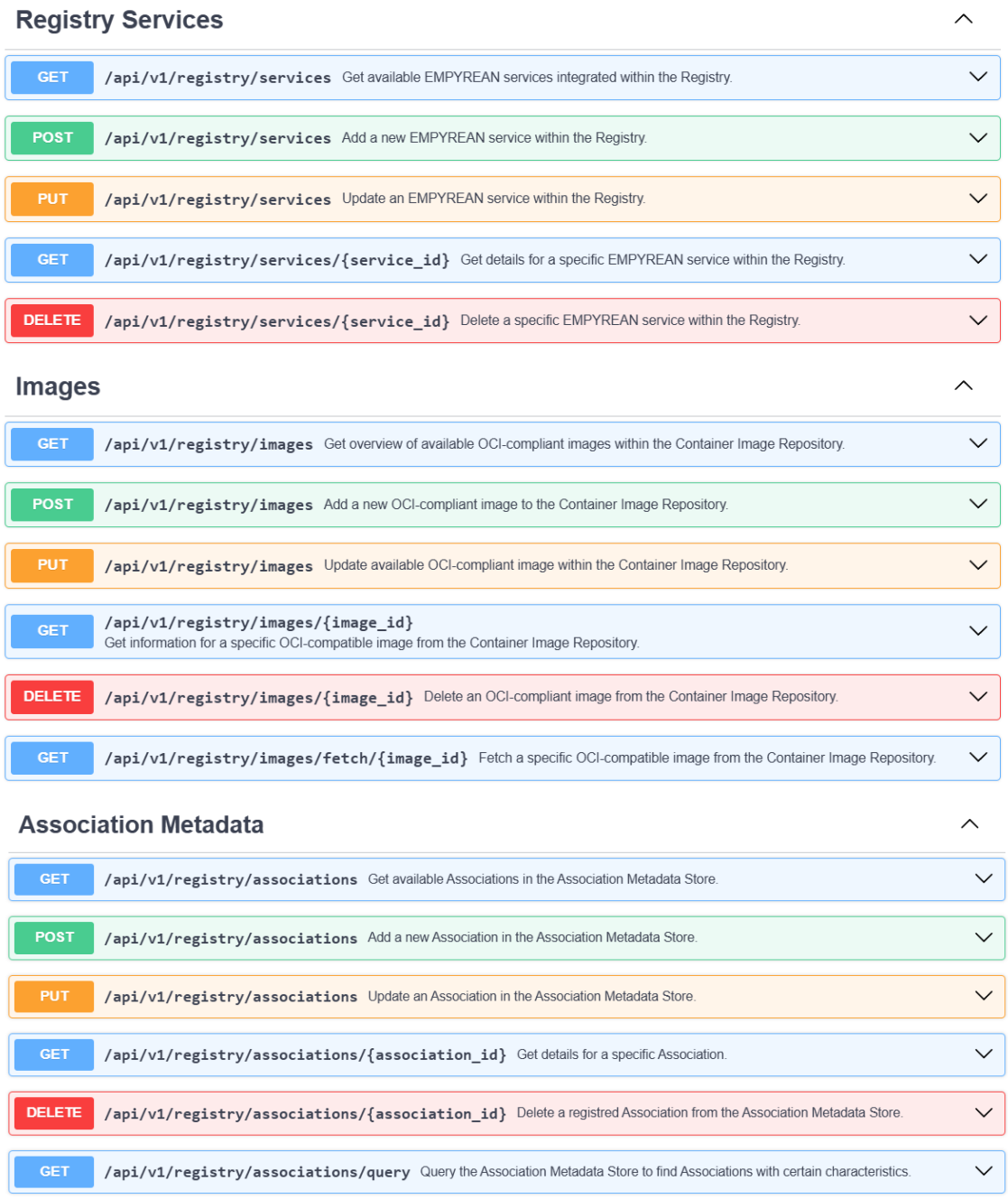
The *Data Connectors* component enables integration with external data sources, facilitating the collection and exchange of data across heterogeneous environments. It is designed with a modular and extensible framework, allowing easy addition of new connectors. This component supports the integration with data stores, external catalogues, data pipelines, and real-time data streams. By providing structured metadata ingestion and aggregation, Data Connectors enhance the context-awareness and adaptability of the EMPYREAN Registry.

Moreover, the integration of *Security and Trust Management* services establishes a trust anchor to support trust, identity, and credential management operations across the distributed Associations. This ensures secure interactions, enhances reliability, and promotes seamless collaboration within the EMPYREAN ecosystem. A detailed presentation of this service's design and initial implementation within the EMPYREAN platform is available in deliverable D3.1 (M15).

11.4 Implementation

The EMPYREAN Registry is implemented as a cloud-native application in Python, where individual components interact through their well-defined APIs. All components of the Registry are containerized and orchestrated via Kubernetes, allowing for easy deployment, dynamic scaling, and fault tolerance. The work during the initial phase covers the initial design and implementation of the API Gateway, Association Metadata Store, and Container Image Repository components.

The API Gateway exposes a *REST interface* to facilitate stateless communication and provide external access to Registry services and low-complexity interactions. There is also a *gRPC* interface to support low-latency and bidirectional streaming event-driven interactions and real-time updates, which is well-suited for service-to-service communication among the core components of the EMPYREAN control and management plane. Both interfaces rely on the Registry Manager, who handles the requests and interacts with the other internal components. Figure 43 shows the available methods in the REST interfaces of the API Gateway component.



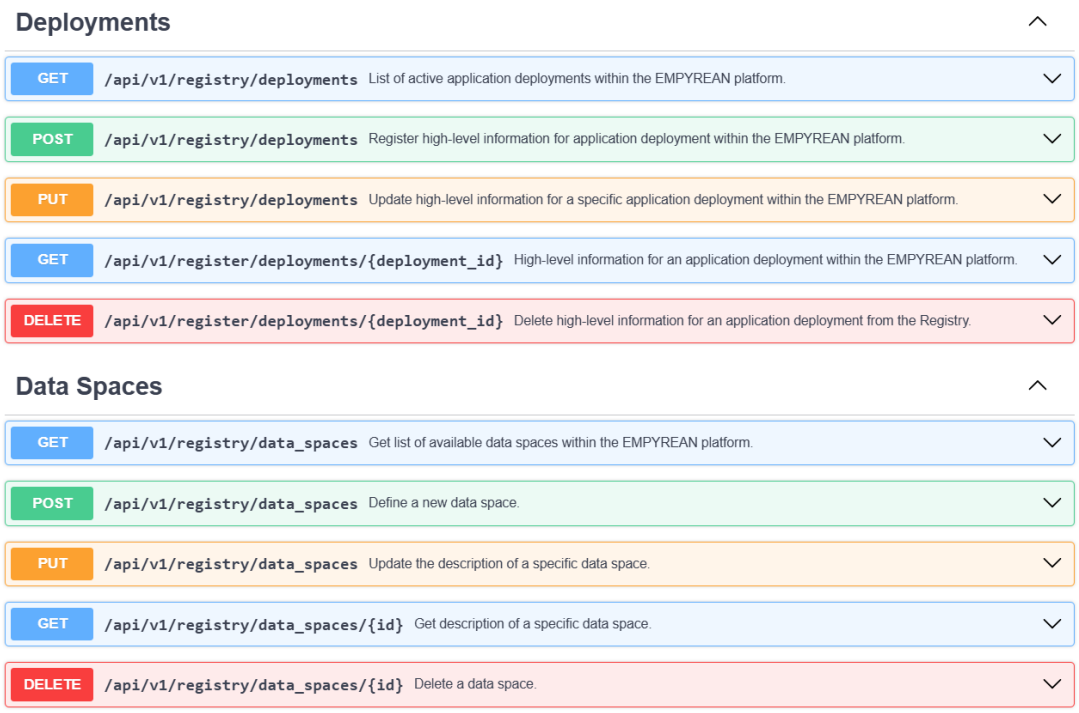


Figure 43: EMPYREAN Registry – API Gateway RESTful API.

Table 9 presents the RPC methods exposed by the initial version of Registry's gRPC interface, covering the communication between Registry and Aggregator for the Associations management.

Table 9: EMPYREAN Registry – RPC methods.

Method Name	Description
AssociationUpdates	Provide real-time updates to the EMPYREAN Registry for available Associations in the overall platform.
ServicesUpdates	Provide real-time updates to the EMPYREAN Registry for available services in the overall platform.
DeploymentUpdates	Provide real-time updates to the EMPYREAN Registry for available application deployments in the platform.

The Container Image Repository is based on the CNCF Distribution Registry[16], an open-source stateless and highly scalable storage and content delivery system that holds named container images and other content, available in different tagged versions. One key feature is that it adheres to OCI specifications for storing and retrieving container images, Helm charts, WASM artifacts, and other OCI-based artifacts, enabling seamless integration with the EMPYREAN build and package mechanisms. Moreover, it is a decentralized and extensible solution that integrates well with Kubernetes and other cloud-native tools while can be deployed across various cloud and edge environments.

[16] https://distribution.github.io/distribution/

The Association Metadata Store constitutes a fundamental element of the EMPYREAN ecosystem, consolidating metadata from multiple distributed Associations. Its primary role is to maintain structured information about each Association's participating resources, workload, sharing and other policies, enabling coordination, trust, and intelligent orchestration across the Association-based continuum.

Each Association operates independently, orchestrating tasks and managing the computing infrastructure. Static and dynamic information is collected — including the current state of nodes, Kubernetes pods, deployments, and workflow executions. This metadata will be used to construct a local Knowledge Graph (KG) for each Association, using a graph database such as Neo4j[17]. The resulting graph will reflect the relationships between computational entities and infrastructure components, capturing how services are deployed, which nodes they run on, and how they are managed. Moreover, these graphs will be periodically merged to create an aggregated Knowledge Graph, maintained in the Association Metadata Store. This merged graph will offer a unified semantic representation of the distributed state of the continuum. One of the key benefits of this approach is the ability to run cross-Association queries over the summarized graph. These queries enable rich, relationship-based reasoning — such as identifying trends across multiple workloads, detecting anomalies in pods or resource usage, or coordinating deployments to avoid infrastructure hotspots. The graph-based representation is especially powerful in scenarios where relationships between resources, services, and infrastructure need to be continuously monitored and optimized.

11.5 Relation to Use Cases

The EMPYREAN Registry plays a central role in enabling dynamic, secure, and efficient resource and service management across all project use cases (UCs). By serving as a unified point for discovering, registering, and managing Associations, services, and infrastructure resources, it facilitates the deployment and orchestration of applications tailored to the specific requirements of each domain. Its core functionalities—including metadata aggregation, service cataloguing, and container image distribution—directly support the underlying Association-based continuum of each use case.

The Registry enables the registration and discovery of IoT devices, robotic systems, edge, and cloud resources deployed across diverse industrial and agricultural environments. The Service Catalogue and Association Metadata Store facilitate context-aware workload placement by tracking the ownership and sharing policies of connected resources, enabling privacy-preserving computation and intelligent orchestration. These capabilities ensure resilient service deployment in resource-constrained, highly distributed environments. By integrating with the Security and Trust Management services, the Registry ensures secure service registration, identity verification, and access control, providing a secured and trustworthy execution environment across collaborated geographically distributed resources.

[17] https://neo4j.com

12 Conclusions

In this deliverable, we present the work of all tasks in WP4, with a particular focus on Task 4.1 *"Cyber Threat Intelligence, Intelligent Resource Management and Energy Efficiency"* and Task 4.4 *"EMPYREAN Aggregator, Autonomous Management and Monitoring Fabric"*. Specifically, we elaborate on the design and developments for: (i) intelligent and multi-objective resource management algorithms, (ii) Decision Engine, (iii) cyber threat intelligent platform, (iv) Service Orchestrator and EMPYREAN Controller, (v) Telemetry Service, (vi) EMPYREAN Aggregator, and (viii) EMPYREAN Registry. Together, these components form a cohesive platform designed to meet the demands of next-generation applications, providing resilience, intelligence, adaptability, and trustworthiness in highly dynamic, resource-constrained, and distributed operating environments. They also enable collaborative autonomy and support the cooperative and autonomous management of the Association-based continuum, also promoting self-driven adaptability.

The provided developments are integral parts of the autonomous and cognitive control and management plane of the EMPYREAN platform. The presented research and development activities build upon the final architecture of the EMPYREAN, reported in D2.3 (M12), to provide the core functionality, implement the initial version of interfaces for inter-component communication, and support the implementation of the basic operation flows. The above developments will be further enhanced as we move towards the second iteration of the implementation plan (M16-M26) to fully support the envisioned functionalities within the complete release of the EMPYREAN integrated platform.

13 References

[1] Atieh, A. T. (2021). The next generation cloud technologies: A review on distributed cloud, fog and edge computing and their opportunities and challenges. Pages 1–15.

[2] Auer, F., Lenarduzzi, V., Felderer, M., and Taibi, D. (2021). From monolithic systems to microservices: An assessment framework. Information and Software Technology, 137:106600.

[3] Chen, S., Delimitrou, C., and Mart´ınez, J. F. (2019). Parties: Qos-aware resource partitioning for multiple interactive services. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19, pages 107–120, New York, NY, USA. ACM.

[4] Convolbo, M. W. and Chou, J. (2016). Cost-aware dag scheduling algorithms for minimizing execution cost on cloud resources. The Journal of Supercomputing, 72(3):985–1012.

[5] Dally, W. J., Turakhia, Y., and Han, S. (2020). Domainspecific hardware accelerators. Communications of the ACM, 63(7):48–57.

[6] Dzhagaryan, A. and Milenkovic, A. (2014). Impact of ´ thread and frequency scaling on performance and energy in modern multicores. In Proceedings of the 52nd ACM Southeast Conference (ACM SE '14), pages Article 30, 6 pages, New York, NY, USA. ACM.

[7] Einav, Y. (2023). Amazon found every 100ms of latency cost them 1. Accessed: 2024-11-05.

[8] Garcia, A. M., Serpa, M., Griebler, D., Schepke, C., ao, L. L., and Navaux, P. O. A. (2020). The impact of CPU frequency scaling on power consumption of computing infrastructures. In High Performance Computing. SBAC-PAD 2019 International Workshops, volume 12083 of Lecture Notes in Computer Science, pages 142–157, Cham, Switzerland. Springer.

[9] Gluck, A. (2020). Introducing domain-oriented microservice architecture. Accessed: 2024-11-05.

[10] Gupta, R. and et al. (2021). 6g-enabled edge intelligence for ultra-reliable low latency applications: Vision and mission. Computer Standards & Interfaces, 77:103521.

[11] Hua, W., Liu, P., and Huang, L. (2023). Energy-efficient resource allocation for heterogeneous edge-cloud computing. IEEE Internet of Things Journal, pages 1–1.

[12] Li, Z. and Zhu, Q. (2020). Genetic algorithm-based optimization of offloading and resource allocation in mobile-edge computing. Information, 11(2):83.

[13] Luo, S. and et al. (2022). An in-depth study of microservice call graph and runtime performance. IEEE Transactions on Parallel and Distributed Systems, 33(12):3901–3914.

[14] Mauro, T. (2015). Adopting microservices at netflix: Lessons for architectural design.

[15] NVIDIA (2024). Power management guide for jetson xavier. https://docs.nvidia.com/jetson/. Accessed: 2024-11-17.

[16] Papadimitriou, G., Chatzidimitriou, A., and Gizopoulos, D. (2019). Adaptive voltage/frequency scaling and core allocation for balanced energy and performance on multicore cpus. In Proceedings of the 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 133–144, Washington, DC, USA. IEEE.

[17] Qiu, H., Banerjee, S. S., Jha, S., Kalbarczyk, Z., and Iyer, R. K. (2020). Firm: An intelligent fine-grained resource management framework for slo-oriented microservices.

[18] Somashekar, G. and et al. (2022). Reducing the tail latency of microservices applications via optimal configuration tuning.

[19] Song, C. and et al. (2023). Chainsformer: A chain latencyaware resource provisioning approach for microservices cluster. arXiv preprint arXiv:2309.12592.

[20] Syed, A. (2021). Intel 11th gen rocket lake-s cpu power consumption explained. https://hardwaretimes.com. Accessed: 2024-11-17.

[21] Wikipedia contributors (2021). List of intel processors. https://en.wikipedia.org/wiki/List_of_Intel_processors. Accessed: 2024-11-17.

[22] Zhang, Z., Ramanathan, M. K., Raj, P., Parwal, A., Sherwood, T., and Chabbi, M. (2022). Crisp: Critical path analysis of large-scale microservice architectures. In 2022 USENIX Annual Technical Conference (USENIX ATC 2022), pages 655–672, Carlsbad, CA, USA. USENIX Association.

[23] Zidar, J., Matic, T., Aleksi ´ c, I., and ´ Z. Hocenski (2024). ˇ Dynamic voltage and frequency scaling as a method for reducing energy consumption in ultra-low-power embedded systems. Electronics, 13(5):826.

[24] H. Tianfield, “Towards Edge-Cloud Computing,” Dec. 2018.

[25] S.-H. Chiang, A. C. Arpaci-Dusseau, and M. K. Vernon, “The Impact of More Accurate Requested Runtimes on Production Job Scheduling Performance,” Lecture notes in computer science, pp. 103–127, 2002.

[26] M. Trenz *et al*,, “The Role Of Uncertainty In Cloud Computing Continuance: Antecedents, Mitigators, And Consequences,” 2013.

[27] D. Zhang *et al*, “RLScheduler: An Automated HPC Batch Job Scheduler Using RL,” arXiv (Cornell University), Nov. 2020.

[28] Y. Jin, Y. Wen, Q. Chen, and Z. Zhu, “An Empirical Investigation of the Impact of Server Virtualization on Energy Efficiency for Green Data Center,” The Computer Journal, vol. 56, no. 8, pp. 977–990, 2013.

[29] J. Panneerselvam, L. Liu, J. Hardy, and N. Antonopoulos, “Analysis, Modelling and Characterisation of Zombie Servers in Large-Scale Cloud Datacentres,” IEEE Access, vol. 5, pp. 15040–15054, 2017.

[30] L. Kidane, P. Townend, T. Metsch, and E. Elmroth, “When and How to Retrain Machine Learning-based Cloud Management Systems,” *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2022.

[31] A. Tchernykh, *et al*, “Towards Understanding Uncertainty in Cloud Computing Resource Provisioning,” *Procedia Computer Science*, vol. 51, pp. 1772–1781, 2015.

[32] F. Black and M. Scholes, “The Pricing of Options and Corporate Liabilities,” *Journal of Political Economy*, vol. 81, pp. 637–654, 1973

[33] E. Liu, X. Deng, Z. Cao, and H. Zhang, “Design and Evaluation of a Prediction-Based Dynamic Edge Computing System,”, Dec. 2018,

[34] B. Liu, J. Guo, C. Li, and Y. Luo, “Workload forecasting based elastic resource management in edge cloud,” *Computers & Industrial Engineering*, vol. 139, p. 106136, Jan. 2020.

[35] Z. Qin, *et al*, “Optimal Workload Allocation for Edge Computing Network Using Application Prediction,” Wireless Communications and Mobile Computing, vol. 2021, pp. 1–13, Mar. 2021.

[36] S. Arbat *et al*, “Wasserstein Adversarial Transformer for Cloud Workload Prediction,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, no. 11, pp. 12433–12439, Jun. 2022.

[37] S. Banerjee, S. Roy, and S. Khatua, “Efficient resource utilization using multi-step-ahead workload prediction technique in cloud,” The Journal of Supercomputing, vol. 77, no. 9, pp. 10636–10663, Mar. 2021.

[38] “Horizontal Pod Autoscaler” *Kubernetes* kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale

[39] K. Fu et al, "Adaptive Resource Efficient Microservice Deployment in Cloud-Edge Continuum," in IEEE Transactions on Parallel and Distributed Systems, vol. 33, no. 8, pp. 1825-1840, 2022.

[40] D. Saxena and A. K. Singh, “Workload Forecasting and Resource Management Models based on Machine Learning for Cloud Computing Environments.,” Jun. 2021.

[41] G. Kontos, P. Soumplis, P. Kokkinos, and E. Varvarigos, “Cloud-Native Applications’ Workload Placement over the Edge-Cloud Continuum,” Jan. 2023.

[42] Y. Zhang *et al*, “Multi-agent deep reinforcement learning for online request scheduling in edge cooperation networks,” *Future Generation Computer Systems*, vol. 141, pp. 258–268, Apr. 2023,

[43] Y. Hao *et al*, “EdgeTimer: Adaptive Multi-Timescale Scheduling in Mobile Edge Computing with Deep Reinforcement Learning,” *arXiv (Cornell University)*, pp. 671–680, May 2024.

[44] I. Ullah, *et al*, "Optimizing task offloading and resource allocation in edge-cloud networks: a DRL approach," Journal of Cloud Computing, vol. 12, no. 1, Jul. 2023

[45] D. Reiswich and U. Wystup, "A Guide to FX Options Quoting Conventions," Journal of Derivatives, vol. 18, no. 2, pp. 58–68, 2010.

[46] "Alibaba Cluster Trace Data". https://github.com/alibaba/clusterdata

[47] P. Osypanka and P. Nawrocki, "Resource Usage Cost Optimization in Cloud Computing Using Machine Learning," IEEE Transactions on Cloud Computing, 2020.

[48] W. Jiang, D. Lee, and S. Hu, "Cloud Resource Demand Prediction using Machine Learning in the Context of QoS Parameters," Journal of Grid Computing, 2022.

[49] Xiong Xiong, "Resource Allocation Based on Deep Reinforcement Learning in IoT Edge Computing," IEEE Journal on Selected Areas in Communications, vol. 38, pp. 1133–1146, 2020, doi: 10.1109/JSAC.2020.2986615.

[50] H. Qadeer and D. Lee, "Deep-Deterministic Policy Gradient-Based Multi-Resource Allocation Framework for IoT Edge Computing," IEEE Access, vol. 11, pp. 45033–45046, 2023, doi: 10.1109/ACCESS.2023.3069991.

[51] K. Mitropoulou, P. Kokkinos, P. Soumplis, and E. Varvarigos, "Anomaly detection in cloud computing using knowledge graph embedding and machine learning mechanisms," Journal of Grid Computing, vol. 22, no. 1, p. 6, 2024.

[52] K. Mitropoulou, P. Kokkinos, P. Soumplis, and E. Varvarigos, "Detect resource related events in a cloud-edge infrastructure using knowledge graph embeddings and machine learning," in 2022 13th International Symposium on Communication Systems, Networks and Digital Signal Processing (CSNDSP), 2022, pp. 698–703.

[53] Zhu Tianqing, "Resource Allocation in IoT Edge Computing via Concurrent Federated Reinforcement Learning," IEEE Internet of Things Journal, vol. 9, pp. 1414–1426, 2021, doi: 10.1109/JIOT.2021.3086910.

[54] J. Liu and others, "Optimized Task Allocation for IoT Application in Mobile-Edge Computing," IEEE Internet of Things Journal, vol. 9, no. 12, pp. 10370–10381, 2022, doi: 10.1109/JIOT.2021.3091599.

[55] Y. Zhu, Y. Gao, and Y. Tong, "A Survey of Federated Learning for Edge Computing: Research Problems and Solutions," IEEE Transactions on Industrial Informatics, vol. 17, no. 4, pp. 2143–2156, 2021, doi: 10.1109/TII.2020.3037875.

[56] G. L. Xing Chen, "Federated Deep Reinforcement Learning-Based Task Offloading and Resource Allocation for Smart Cities in a Mobile Edge Network," Sensors (Basel, Switzerland), vol. 22, p. 4738, 2022, doi: 10.3390/s22134738.

[57] S. Lingayya et al., "Dynamic Task Offloading for Resource Allocation and Privacy-Preserving Framework in KubeEdge-Based Edge Computing Using Machine Learning," Cluster Computing, 2024.

[58] S. Tian et al., "Blockchain-Based 6G Task Offloading and Cooperative Computing Resource Allocation Study," Journal of Cloud Computing, 2024.

[59] A. Dandoush, A. Gouissem, and B. S. Ciftler, "Secure and Privacy-Preserving Federated Learning-Based Resource Allocation for Next Generation Networks," IEEE Transactions on Network and Service Management, 2023.

[60] Y. Zhan and J. Zhang, "An incentive mechanism design for efficient edge learning by deep reinforcement learning approach," IEEE INFOCOM, pp. 2489–2498, 2020.

[61] Y. Sarikaya and O. Ercetin, "Motivating Workers in Federated Learning: A Stackelberg Game Perspective," IEEE Networking Letters, vol. 2, no. 1, pp. 23–27, 2019.

[62] C. Yang, M. Xu, and Q. Wang, "FLASH: Heterogeneity-Aware Federated Learning at Scale," IEEE Transactions on Mobile Computing, 2022.

[63] S. Yu, X. Chen, and Z. Zhou, "Experience-Driven Computational Resource Allocation of Federated Learning by Deep Reinforcement Learning," IEEE International Parallel and Distributed Processing Symposium, pp. 234–243, 2021.

[64] Syed, Z., Padia, A., Mathews, M. L., Finin, T., & Joshi, A. (2016, February). UCO: A unified cybersecurity ontology. In Proceedings of the AAAI Workshop on Artificial Intelligence for Cyber Security (pp. 195-202).

[65] Eunsoo Kim, Kuyju Kim, Dongsoon Shin, Beomjin Jin, and Hyoungshick Kim. 2018. CyTIME: Cyber Threat Intelligence ManagEment framework for automatically generating security rules. In Proceedings of the 13th International Conference on Future Internet Technologies (CFI 2018). Association for Computing Machinery, New York, NY, USA, Article 7, 1–5. https://doi.org/10.1145/3226052.3226056

[66] Y. Gao, X. Li, H. Peng, B. Fang and P. S. Yu, "HinCTI: A Cyber Threat Intelligence Modeling and Identification System Based on Heterogeneous Information Network," in IEEE Transactions on Knowledge and Data Engineering, vol. 34, no. 2, pp. 708-722, 1 Feb. 2022, doi: 10.1109/TKDE.2020.2987019.

[67] Ibrahim Alabdulmohsin, YuFei Han, Yun Shen, and Xiangliang Zhang. 2016. Content-agnostic malware detection in heterogeneous malicious distribution graph. In Proceedings of the 25th ACM International on Conference on Information and Knowledge Management. 2395–2400.

[68] Mahathir Almashor, Ejaz Ahmed, Benjamin Pick, Sharif Abuadbba, Jason Xue, Raj Gaire, Shuo Wang, Seyit Camtepe, and Surya Nepal. 2022. Unraveling Threat Intelligence Through the Lens of Malicious URL Campaigns. arXiv preprint arXiv:2208.12449 (2022).

[69] Eihal Alowaisheq, Siyuan Tang, Zhihao Wang, Fatemah Alharbi, Xiaojing Liao, and XiaoFeng Wang. 2020. Zombie Awakening: Stealthy Hijacking of Active Domains through DNS Hosting Referral. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (Virtual Event, USA) (CCS '20). Association for Computing Machinery, New York, NY, USA, 1307–1322. https://doi.org/10.1145/3372297.3417864

[70] Xander Bouwman, Harm Griffioen, Jelle Egbers, Christian Doerr, Bram Klievink, and Michel van Eeten. 2020. A different cup of TI? The added value of commercial threat intelligence. In 29th USENIX Security Symposium (USENIX Security 20). USENIX Association, 433–450. https://www.usenix.org/conference/usenixsecurity20/presentation/bouwman

[71] Xander Bouwman, Victor Le Pochat, Pawel Foremski, Tom Van Goethem, Carlos H. Ganan, Giovane C. M. Moura, Samaneh Tajalizadehkhoob, Wouter Joosen, and Michel van Eeten. 2022. Helping hands: Measuring the impact of a large threat intelligence sharing community. In the 31st USENIX Security Symposium (USENIX Security 22). USENIX Association, Boston, MA, 1149–1165. https://www.usenix.org/conference/usenixsecurity22/presentation/bouwman

[72] Birhanu Eshete and V. N. Venkatakrishnan. 2014. WebWinnow: Leveraging Exploit Kit Workflows to Detect Malicious Urls. In Proceedings of the 4th ACM Conference on Data and Application Security and Privacy (San Antonio, Texas, USA) (CODASPY '14). Association for Computing Machinery, New York, NY, USA, 305–312. https://doi.org/10.1145/2557547.2557575

[73] Siwon Huh, Seonghwan Cho, Jinho Choi, Seungwon Shin, and Hojoon Lee. 2022. A Comprehensive Analysis of Today's Malware and Its Distribution Network: Common Adversary Strategies and Implications. IEEE Access 10 (2022), 49566–49584.

[74] Colin C Ife, Yun Shen, Steven J Murdoch, and Gianluca Stringhini. 2019. Waves of malice: A longitudinal measurement of the malicious file delivery ecosystem on the web. In Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security. 168–180.

[75] Colin C Ife, Yun Shen, Steven J Murdoch, and Gianluca Stringhini. 2021. Marked for disruption: tracing the evolution of malware delivery operations targeted for takedown. In 24th International Symposium on Research in Attacks, Intrusions and Defenses. 340–353.

[76] Luca Invernizzi, Stanislav Miskovic, Rubén Torres, Christopher Krügel, Sabyasachi Saha, Giovanni Vigna, Sung-Ju Lee, and Marco Mellia. 2014. Nazca: Detecting Malware Distribution in Large-Scale Networks. In Network and Distributed System Security Symposium.

[77] Dohoon Kim. 2019. Potential Risk Analysis Method for Malware Distribution Networks. IEEE Access 7 (2019), 185157–185167. https://doi.org/10.1109/ACCESS.2019.2960552

[78] Bum Jun Kwon, Jayanta Mondal, Jiyong Jang, Leyla Bilge, and Tudor Dumitras. 2015. The dropper effect: Insights into malware distribution with downloader graph analytics. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. 1118–1129.

[79] Fran¸cois Labrèche, Enrico Mariconti, and Gianluca Stringhini. 2022. Shedding Light on the Targeted Victim Profiles of Malicious Downloaders. In Proceedings of the 17th International Conference on Availability, Reliability and Security. 1–10.

[80] Matthew Luckie, Alexander Marder, Marianne Fletcher, Bradley Huffaker, and K. Claffy. 2020. Learning to Extract and Use ASNs in Hostnames. In Proceedings of the ACM Internet Measurement Conference (Virtual Event, USA) (IMC '20). Association for Computing Machinery, New York, NY, USA, 386–392. https://doi.org/10.1145/3419394.3423639

[81] Adrian Stefan Popescu, Dumitru Bogdan Prelipcean, and Dragos Teodor Gavrilut. 2015. A Study on Techniques for Proactively Identifying Malicious URLs. In 2015 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC). 204–211. https://doi.org/10.1109/SYNASC.2015.40

[82] Noëlle Rakotondravony, Benjamin Taubmann, Waseem Mandarawi, Eva Weish¨aupl, Peng Xu, Bojan Kolosnjaji, Mykolai Protsenko, Hermann De Meer, and Hans P Reiser. 2017. Classifying malware attacks in IaaS cloud environments. Journal of Cloud Computing 6, 1 (2017), 1–12.

[83] Sayak Saha Roy, Unique Karanjit, and Shirin Nilizadeh. 2021. What remains uncaught?: Characterizing sparsely detected malicious urls on twitter.

[84] Yun Shen and Gianluca Stringhini. 2019. ATTACK2VEC: Leveraging Temporal Word Embeddings to Understand the Evolution of Cyberattacks. In 28th USENIX Security Symposium (USENIX Security 19). USENIX Association, Santa Clara, CA, 905–921.

[85] Yun Shen and Gianluca Stringhini. 2021. ANDRUSPEX: leveraging graph representation learning to predict harmful app installations on mobile devices. In 2021 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE, 562–577.

[86] Yun Shen, Pierre-Antoine Vervier, and Gianluca Stringhini. 2021. Understanding worldwide private information collection on android. arXiv preprint arXiv:2102.12869 (2021).

[87] Kevin van Liebergen, Juan Caballero, Platon Kotzias, and Chris Gates. 2022. A Deep Dive into VirusTotal: Characterizing and Clustering a Massive File Feed. arXiv:2210.15973 [cs.CR]

[88] Gang Wang, Jack W. Stokes, Cormac Herley, and David Felstead. 2013. Detecting malicious landing pages in Malware Distribution Networks. In 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). 1–11. https://doi.org/10.1109/DSN.2013.6575316

[89] Mingxuan Yao, Jonathan Fuller, Ranjita Pai Sridhar, Saumya Agarwal, Amit K. Sikder, and Brendan Saltaformaggio. 2023. Hiding in Plain Sight: An Empirical Study of Web Application Abuse in Malware. In USENIX Security Symposium.

[90] Zijun Yao, Yifan Sun, Weicong Ding, Nikhil Rao, and Hui Xiong. 2018. Dynamic word embeddings for evolving semantic discovery. In Proceedings of the eleventh acm international conference on web search and data mining. 673–681.

[91] Shuofei Zhu, Jianjun Shi, Limin Yang, Boqin Qin, Ziyi Zhang, Linhai Song, and Gang Wang. 2020. Measuring and Modeling the Label Dynamics of Online {Anti-Malware} Engines. In 29th USENIX Security Symposium (USENIX Security 20). 2361–2378.