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

Grant Agreement no. 101136024

Deliverable D5.2
Initial release of EMPYREAN integrated platform

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:	WP5
Responsible Editor:	ZSCALE
Due date:	31/07/2025
Actual submission date:	04/08/2025

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
30.04.2025	Ivan Paez (ZSCALE)	Initial Draft	0.1	ToC definition, section definitions
30.05.2025	All partners	Draft	0.2	First round of input collection
30.06.2025	All partners	Complete deliverable	0.5	First complete deliverable
15.07.2025	Reviewers feedback	Reviewers feedback	0.8	Reviewers provide feedback
25.07.2025	Ivan Paez (ZSCALE), main editor	Final review, project coordinator	0.9	Addressing the reviewers' feedback, comments,and corrections
30.07.2025	Ivan Paez (ZSCALE)	Final version	1.0	Format and camera-ready deliverable.

Author List

Organization	Author
CC	Marton Sipos
ICCS	Aristotelis Kretsis, Panagiotis Kokkinos, Fotis Kouzinos, Polizois Soumplis, Emmanouel Varvarigos
NEC	Jaime Fúster, Roberto González
NUBIS	Anastassios Nanos, Christos Panagiotou, Charalampos Mainas, Georgios Ntoutsos, Panagiotis Mavrikos, Maria Gkeka, Maria Gkoutha, Ilias Lagomatis, Apostolos Giannousas
NVIDIA	Dimitris Syrivelis
RYAX	Pedro Velho, Yuqiang Ma, Michael Mercier, Yiannis Georgiou
UMU	Antonio Skarmeta, Eduardo Cánovas, Alonso Sánchez, José Luis Sánchez
ZSCALE	Ivan Paez, Mahmoud Mazouz, Phani Gangula

Internal Reviewers

Aristotelis Kretsis, Polizois Soumplis (ICCS)

Anastassios Nanos (NUBIS)

Abstract: This deliverable summarizes the initial software release of the EMPYREAN platform, which corresponds to the outcomes of T5.1 Integration, Testing, and Refinement from the WP5 Platform Integration and Use Case Development. This deliverable presents the initial platform release, along with a breakdown of the platform's key building blocks and their interactions.

Keywords: EMPYREAN platform, Association, integrated components, development and integration environment, software deployment, operation flows

Table of Contents

List of Figures

List of Tables

Abbreviations

ABAC	Attribute-Based Access Control
AI	Artificial Intelligence
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
CI/CD	Continuous Integration and Continuous Deployment
CLI	Command Line Interface
CNCF	Cloud-Native Computing Foundation
CPU	Central Processing Unit
CRI	Container Runtime Interface
CRUD	Create, Read, Update, Delete
CTA	Cyber Threat Alliance
CTI	Cyber Threat Intelligence
D	Deliverable
DAG	Directed Acyclic Graphs
DICE	Device Identifier Composition Engine
DID	Decentralized Identifier
DLT	Distributed Ledger Technology
ESS	Edge Storage Service
FIFO	Fist-In First-Out
GDD	Generalized Data Deduplication
GHCR	GitHub Container Registry
GPU	Graphics Processing Unit
I/O	Input / Output
IAM	Identity and Access Management
IoC	Indicator of Compromise
IoT	Internet of Things
JWT	JSON Web Tokens
K3s	Lightweight Kubernetes
K8s	Kubernetes
KPI	Key Performance Indicator
M	Month
MIG	Multiple-Instance GPU
MISP	Malware Information Sharing Platform
ML	Machine Learning
MQTT	Message Queue Telemetry Transport
NBI	Northbound Interface
NIC	Network Interface Card
OF	Operation Flow
ORM	Object-Relational Mapping
OTEL	OpenTelemetry
p-ABC	Privacy-Preserving Attribute-Based Credential
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PSM	Privacy and Security Manager
RDMA	Remote Direct Memory Access

REST	REpresentational State Transfer
RLNC	Random Linear Network Coding
RNIC	RDMA NIC
TCP	Transmission Control Protocol
TEE	Trusted Execution Environment
TRL	Technology Readiness Level
UI	User Interface
UUID	Universally Unique Identifier
VC	Verifiable Credential
VP	Verifiable Presentation
WP	Work Package
XACML	eXtensible Access Control Markup Language
ZKP	Zero-Knowledge Proofs

1 Executive Summary

The EMPYREAN project aims to address a broad range of challenges that arise in the hyper-distributed computing paradigm, which spans heterogeneous IoT devices and computing, storage, and connectivity resources. These resources may belong to different providers and exist across various segments of the IoT-cloud-edge computing continuum.

EMPYREAN introduces a novel approach centred on federations of collaborative resources and functionalities, referred to as IoT-Edge Associations or simply Associations. These Associations are autonomously created and operated and seamlessly using the EMPYREAN's AI-enabled management framework. Within this Association-based continuum, distributed, cognitive, and dynamic AI-enabled decision-making mechanisms balance computing tasks and data both locally inside an Association as well as between federated Associations in a decentralized, multi-agent setup. The ultimate goal is to optimize resources usage while providing scalability, resiliency, energy efficiency, and quality of service.

EMPYREAN also addresses key challenges including: (i) device volatility and heterogeneity, (ii) virtualization of continuum infrastructure and diverse network connectivity, (iii) optimized and scalable service execution and performance, (iv) efficient resource utilization, including energy usage, (v) trust, security, and privacy guarantees, (vi)reduce integration costs and mitigate vendor lock-in, (vii) promotion of openness, adaptability, and data sharing, and (viii) support for future edge services and data market.

This deliverable describes and accompanies the first iteration release of the EMPYREAN platform. This initial release is based on the work carried out by the consortium initially to define the system's requirements and specifications, reported D2.1 (M6)[1], then to incrementally design the EMPYREAN architecture and targeted operation flows (as described in deliverables D2.2 (M7) [2] and D2.3 (M12), and finally to implement the relevant functionalities within WPs 3 and 4 during the first iteration of the implementation phase (M1-M15). The document outlines the functionalities implemented and integrated so far, targeting a first subset of project objectives and use cases functionalities.

This alpha release of the EMPYREAN platform represents a preliminary implementation of core functionalities, many of which are at the proof-of-concept stage. It enables early integration testing and helps identify gaps along with their respective solutions. It also outlines the roadmap for the upcoming beta and final releases, detailing the development workflow, testbed setup, and integration procedures. To support ongoing development and validation, the project has established a continuous integration and continuous deployment (CI/CD) environment. Using this setup, the initial EMPYREAN release has been delivered, demonstrating core platform functionality and serving as a proof-of-concept for most parts of the architecture.

2 Introduction

2.1 Purpose of this Document

The purpose of D5.2 is to introduce the Continuous Integration (CI) and Continuous Deployment (CD) process carried out in the EMPYREAN project during the preparation of the initial release of its integrated platform. These DevOps practices aim to automate and streamline the software development lifecycle. Here's a breakdown of each and how they work together. The key steps for CI are:

- **Code Commit:** Developers commit code to a shared version control repository.
- **Automated Build:** A CI tool (e.g., Jenkins, GitHub Actions, GitLab CI) automatically builds the code.
- **Automated Testing:** Unit tests and other test suites (e.g., linting, static analysis) are executed.
- **Feedback:** Developers are notified of any errors or failures.

Continuous Delivery and Continuous Deployment (CD) continues from CI and includes:

- **Artifact Packaging**: Artifacts (e.g., Docker images, binaries) are packaged.
- **Staging Environment**: Code is deployed to a staging environment for further testing and validation (e.g., integration and smoke tests).
- **Deployment Approval (optional)**: Manual approval may be required for production deployment, in the case of continuous delivery.
- **Production Deployment**: If all checks pass, the system is deployed to production.
- **Monitoring & Rollback**: Observability tools (e.g., Prometheus, Grafana) monitor the deployment. Rollbacks happen if issues are detected.

2.2 Document Structure

The structure of D5.2 is as follows:

- Section 2 presents the introduction.
- Section 3 provides an overview of EMPYREAN platform and initial release status.
- Section 4 presents the CI&CD process adopted for the EMPYREAN development.
- Section 5 describes the implemented functionalities, testing, and initial integrations for the EMPYREAN platform components.
- Section 6 demonstrates how the components developed and integrated as part of the initial EMPYREAN release support the key operation flows.
- Section 7 concludes the deliverable.

2.3 Audience

This document is intended for software engineers, developers, analysts, and DevOps practitioners. Software architects may find the CI/CD patterns and tooling choices insightful, while end users can gain a better understanding of the complexity and rigor behind deploying a secure and scalable edge-cloud platform like EMPYREAN.

3 The EMPYREAN Platform

3.1 EMPYREAN Architecture

The EMPYREAN architecture was initially introduced in Deliverable D2.2 “Initial Release of EMPYREAN Architecture” (M7) and subsequently refined in its final form in Deliverable D2.3 “Final EMPYREAN architecture, use cases analysis and KPIs” (M12), incorporating feedback from the first iteration of implementation activities. Deliverable D2.3 offers a comprehensive overview of the system architecture, detailing the EMPYREAN components, their interfaces, and the supported operational workflows. In this section, we provide a summary of the architecture (Figure 1) to support the presentation of the initial release of the integrated EMPYREAN platform.

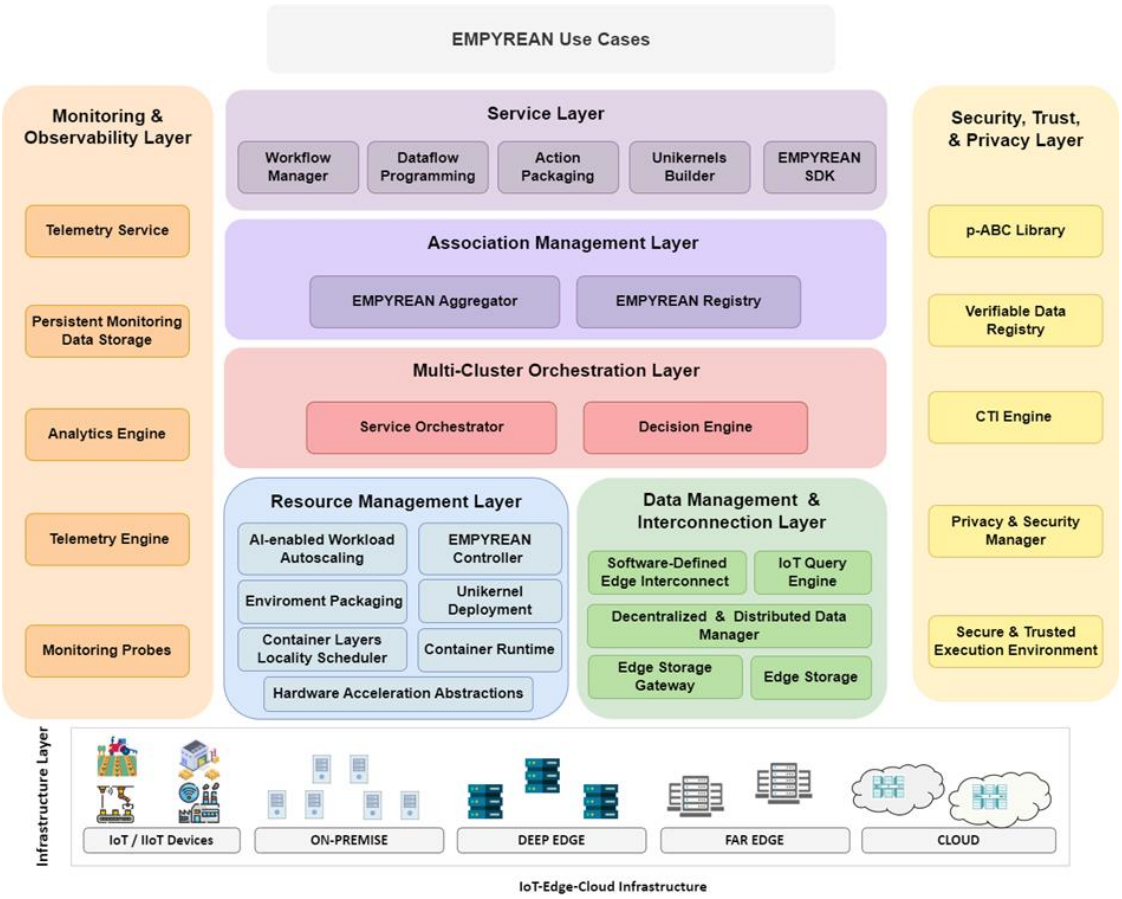


Figure 1: EMPYREAN high-level architecture.

The ***Service Layer*** encompasses components that facilitate the development of Association-native applications, offering robust support for application-level adaptations, interoperability, elasticity, and scalability across the IoT-edge-cloud continuum. This layer focuses on key aspects such as: (a) workflow design and management, simplifying the creation and orchestration of hyper-distributed applications, (b) cloud-native unikernel application development, supporting lightweight, secure, and efficient deployment models,

and (c) dataflow description, enabling precise and scalable data management within applications.

The ***Association Management Layer*** dynamically manages Associations within the IoT-edge-cloud continuum. Forming resource federations enables seamless collaboration, resource sharing, and data distribution across various segments within the continuum. Together with the Multi-Cluster Orchestration Layer, it is central to EMPYREAN's distributed and autonomous management, establishing a resilient Association-based continuum.

The ***Multi-Cluster Orchestration Layer*** handles service orchestration and resource management across EMPYREAN's disaggregated infrastructure. Using autonomous, distributed decision-making mechanisms, it orchestrates dynamic, hyper-distributed applications while enabling self-driven adaptations. Multiple instances of this layer's components provide decentralized operation, optimize resource utilization, and ensure scalability, resiliency, energy efficiency, and high service quality.

The ***Resource Management Layer*** unifies the management of IoT, edge, and cloud platforms under the EMPYREAN platform. It integrates software mechanisms for both platform-level scheduling (e.g., EMPYREAN Controller, AI-enabled Workload Autoscaling) and low-level mechanisms (e.g., Unikernel Deployment). This layer operates within Kubernetes or K3s clusters and offers modularity, simplifying the integration of new hardware and software.

The ***Data Management and Interconnection Layer*** ensures dynamic communication and secure data storage between IoT devices and computing resources. Operating at both cluster and Association levels, it provides flexible and scalable data management and seamless integration of IoT, edge, and cloud resources. It also supports distributed operation, facilitating efficient operation in complex, distributed environments.

The ***Security, Trust, and Privacy Layer*** ensures secure access, privacy, and trusted execution across the EMPYREAN platform. Operating at both the cluster and Association levels, it delivers distributed trust services, enables secure and trusted execution environments, and provides controlled data access for guaranteeing data confidentiality and continuous validation of trust among entities.

Finally, the **Monitoring and Observability Layer** integrates real-time monitoring, observability, and service assurance components to provide full visibility and control over the platform. It uses distributed and automated telemetry mechanisms to dynamically collect diverse metrics from heterogeneous infrastructures and deployed applications. These mechanisms continuously track the health, performance, and availability of IoT devices, edge/cloud infrastructures, platform services, and applications, facilitating data-driven decision-making and enabling advanced automation capabilities.

3.2 Initial Release Status

The initial release of the EMPYREAN platform comprises the components developed during the project's first implementation iteration (M1–M18). This version represents a partial yet functional integration of the platform's architecture, with each component delivering a subset of the targeted features along with the primary interfaces required seamless for inter-component communication.

The primary objective of this release is to deliver a working prototype that demonstrates the core functionalities of the EMPYREAN platform, validating the design choices and enabling early-state testing and feedback collection.

Table 1 provides a high-level overview of the platform components included in this release, along with those scheduled for integration in the second release milestone (M30). A detailed description of the components delivered in this first release is provided in Section 5.

Table 1: Integration status of EMPYREAN components and interfaces.

Layer	Component	Status at First Release (M18)	Second planned Release (M30)
Service Layer	Workflow Manager	✅	
	DataFlow Programming		☑
	Action Packaging	✅	
	Unikernel Building	✅	
	EMPYREAN SDK		☑
Association Management Layer	EMPYREAN Aggregator	✅	
	EMPYREAN Registry	✅	
Multi-cluster Orchestration Layer	Decision Engine	✅	
	Service Orchestrator	✅	
Resource Management Layer	AI-Enabled Workloads Autoscaling	✅	
	EMPYREAN Controller	✅	
	Environment Packaging		☑
	Unikernel Deployment	✅	
	Container Layers Locality Scheduler		☑
	Container Runtime	✅	
	Hardware Acceleration Abstractions	✅	
Data Management and Interconnection Layer	Software Defined Edge Interconnect	✅	
	IoT Query Engine		☑
	Decentralized & Distributed Data Manager	✅	
	Edge Storage & Edge Storage Gateway	✅	

Security, Trust and Privacy Layer	p-ABC Library (UMU) Verifiable Data Registry CTI Engine Privacy & Security Manager Secure & Trusted Execution Environment	✅ ✅ ✅ ✅ ✅	
Monitoring and Observability Layer	Telemetry Service/Engine Persistent Monitoring Data Storage Analytics Engine Monitoring Probes	✅ ✅ ✅	✅

3.3 Integration Infrastructure

EMPYREAN leverages multiple distributed infrastructures, provided by its partners, to deliver the necessary resources supporting integration, qualification, and release processes.

The infrastructure at ICCS offers a versatile testbed located at the HSCNL premises, designed to accommodate a wide range of computing and networking requirements, including virtualization, container orchestration, and secure remote access. It is built around a Proxmox [3] cluster deployed over two dedicated physical servers (DELL PowerEdge R530 and DELL PowerEdge R6515). Within this environment, two distinct clusters have been configured Figure 2 to serve different operational needs: a Kubernetes (K8s) cluster and a Lightweight Kubernetes (K3s) cluster, interconnected via a reconfigurable network setup. This dual-cluster setup enables seamless orchestration of complex services and flexible resource allocation across diverse environments, closely mimicking real-world deployment scenarios.

Figure 2: ICCS testbed – Computation and storage resources at K8s and K3s clusters.

Table 2 summarizes the key characteristics of these clusters. They are utilized for deploying the EMPYREAN control and management plane components, creating Associations, as well as running containerized workloads.

Table 2: K8s and K3s clusters within the ICCS testbed.

Name	Overview	Worker Node Characteristics
K8s	1 x master node 4 x worker nodes	3 x worker nodes on VMs, each one: architecture: amd64 8 CPU cores 16 GB RAM 160 GB DISK 1 x Jetson Orin Nano, each one: architecture: arm64 6 CPU cores 8 GB RAM 64 GB DISK hardware acceleration (GPU)
K3s	1 x master node 3 x worker nodes	2 x Jetson Orin Nano, each one: architecture: arm64 6 CPU cores 8 GB RAM 64 GB DISK hardware acceleration (GPU) 1 x Raspberry Pi 4, each one architecture: arm64 4 CPU cores 8 GB RAM 32 GB DISK

4 Development and Integration Environment

4.1 Continuous Integration (CI/CD) and Continuous Deployment Process

EMPYREAN embraces Continuous Integration and Continuous Deployment (CI/CD) practices to establish a standardized, automated workflow for developing, testing, and releasing its software components. This approach serves two primary goals: (i) to support developers in evaluating and optimizing the performance of their services, and (ii) to address the concerns of service operators when integrating third-party services into their infrastructure. This section highlights the pivotal role of CI/CD in EMPYREAN's development pipeline. Acknowledging their importance, our strategy is carefully designed to improve efficiency, enhance reliability, and accelerate delivery of software components across the platform.

Continuous Deployment (CD) ensures that every validated code change pass through all pipeline and is then deployed into production. Validated features are first tested in a staging environment and then seamlessly promoted to production environment without manual intervention. This process aims to reduce time-to-market, lower deployment risks, and improve operational efficiency by delivering small, incremental updates to end users in a reliable and cost-effective manner.

In a Continuous Integration (CI) environment, developers frequently merge new or modified code into a shared codebase. The CI/CD pipeline embodies a set of practices and tools that enable teams to deliver high-quality code more frequently and reliably. It achieves this by automating the build and testing processes, validating code both locally and at the integration level before it is merged into the mainline. This continuous feedback loop significantly accelerates release cycles, enhances debugging efficiency, and streamlines overall development efforts.

Figure 3 illustrates these pipeline steps that the EMPYREAN platform implements.

1. The first step of the pipeline is when the developers commit their code source to the code source repository[1]. The CI triggers the building process.
2. The CI triggers the run of the internal testing process. If the testing is successful pipeline continues to the next step.
3. Packages are deployed to staging in a Kubernetes environment
4. Packages are also deployed in the Docker registry
5. Deploy to production. this step can be set up automatically or manually.
6. Monitor the application testing/usage by the application developer, alert if there are issues

[1] EMPYREAN - Source code repository in GitHub. https://github.com/empyrean-eu

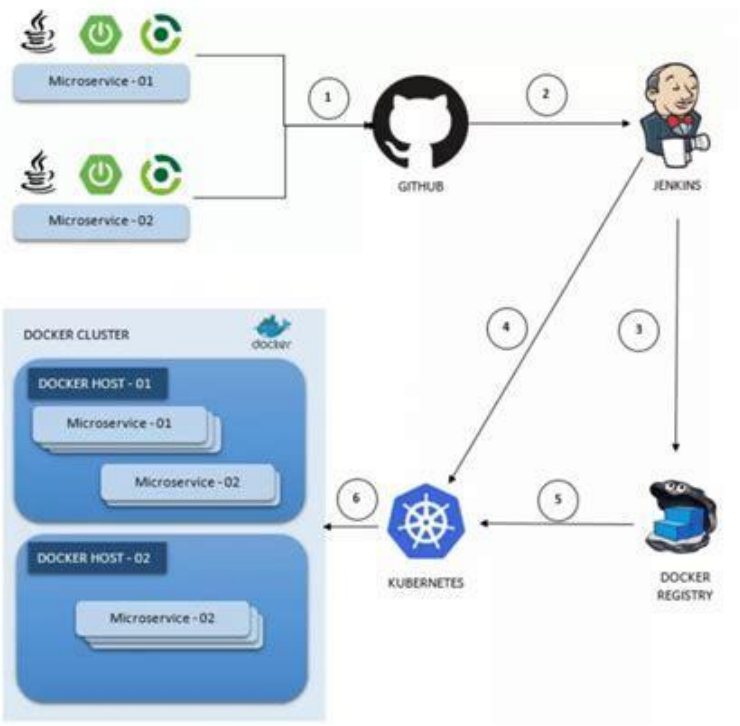


Figure 3: Generic Continuous Development, Continuous Integration (CD/CI) pipeline.

4.1.1 Integration Testing Approaches

For the integration test for the different EMPYREAN components, three different approaches were considered:

- **Top-Down Approach (incremental):** The top-down approach begins integration from the top-level modules and progresses downward. Stubs are used to simulate lower-level modules during early testing. While this approach validates high-level functionality early and can identify major design flaws, it delays testing of lower-level modules and requires the time-consuming development of stubs. Consequently, integration issues in lower-level modules might only surface late in the process.
- **Bottom-Up Approach (incremental):** Conversely, the bottom-up approach starts from the lowest-level modules and moves upward, using drivers to simulate higher-level modules. This method allows for early validation of lower-level functionality, making it easier to identify and fix defects in these modules. However, high-level functionality is tested late, and the development of drivers can also be time-consuming, leading to potential delays in discovering integration issues in higher-level modules.
- **Big-Bang Approach (non-incremental):** The Big-Bang approach involves integrating all modules simultaneously and testing the complete system. This approach offers comprehensive testing of the entire system, faster integration, and immediate identification of interaction issues. It simplifies management by eliminating the need for stubs and drivers. However, debugging can be challenging due to the simultaneous

integration of many components, and identifying the source of defects can be complex. This method requires all modules to be ready for integration simultaneously.

The Big-Bang integration approach has been selected for the EMPYREAN project due to the following key advantages:

- Holistic Validation: EMPYREAN integrates multiple complex components that must function together seamlessly from the beginning.
- Faster Integration: Enables quicker development cycles, essential for meeting project deadlines.
- Immediate Feedback: Allows for rapid identification and resolution of integration issues across components.
- Resource Efficiency: Eliminates the need for stubs and drivers, optimizing resource use.
- Simplified Management: Centralizes integration into a single, manageable phase.
- Best Fit for Tested Modules: Components in EMPYREAN will undergo thorough unit testing, making them suitable for Big-Bang integration.

4.1.2 Integration Workflow

The integration process follows a similar workflow to the generic CI/CD pipeline. EMPYREAN's components integration is automated through GitHub Actions, using the Big-Bang approach to test all modules as a unified system.

Figure 4 illustrates the component's integration pipeline Steps:

1. Code source commit: an EMPYREAN developer pushes code to the project GitHub repository (RYAX), triggering the pipeline.

2. Workflow Trigger: a GitHub Actions workflow is automatically triggered by the commit.

3. Build: the goal is to build an image in the GitHub Runner.

4.1 Code Quality Analysis: the code is analysed using SonarQube to ensure it meets quality standards.

4.2 Unit Testing: individual components are tested in isolation to verify their correctness.

5. Integration Testing: components such as for example the Privacy and Security Manager (PSM) and RYAX Workflow Engine are tested together to validate their interactions within the system.

6a. Publish: if all checks the quality, unit-test, and integration tests pass, a Docker image is built and published to the GitHub Container Registry (GHCR).

6b. Fail Build: if any check fails, the workflow stops and the image is not published.

7. Test Reporting: generates a detailed test report, providing visibility into the status and results of the executed tests.

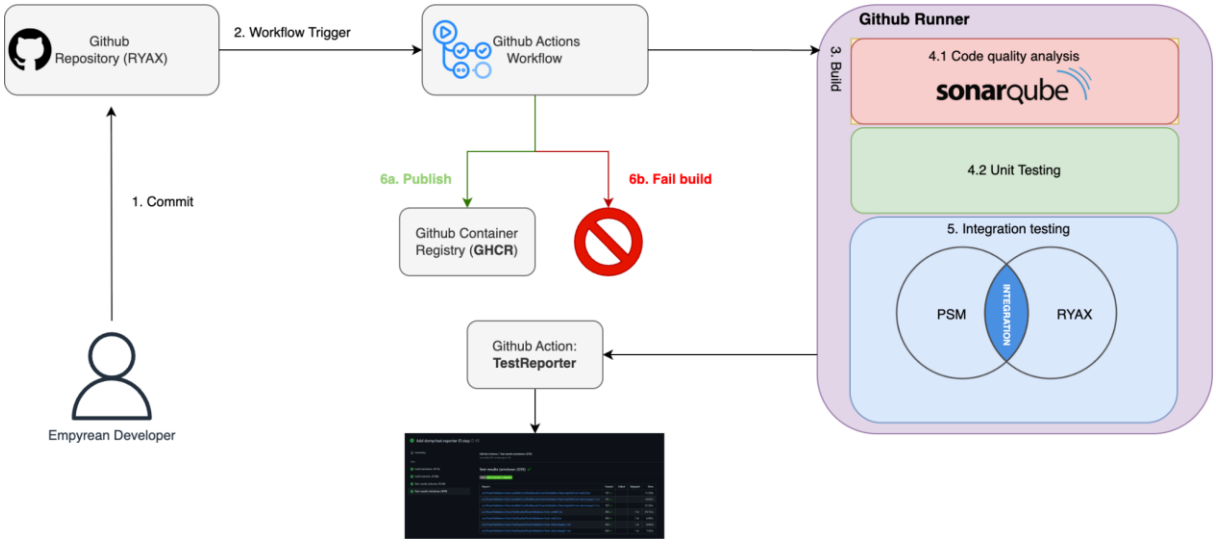


Figure 4: EMPYREAN's components integration workflow.

Integration Requirements for EMPYREAN

Afterwards, to ensure effective integration, the following requirements must be met:

1. **Unit Testing**: each component must include comprehensive unit tests.
2. **Load and Performance Testing**: scripts and guidelines for performance validation must be in place.
3. **Dependency Mapping**: all dependencies (e.g., libraries, tools, frameworks) must be documented per module.
4. **Integration Testing:** designed to verify component interactions, including API calls and service relationships.
5. **Dockerfiles**: each component must have a Docker file for consistent containerization.
6. **Docker-Compose File**: defines multi-container orchestration and service relationships.
7. **Environment Configuration**: clear setup instructions, environment variables, and config files.
8. **API Documentation**: complete and accessible API specs, including endpoints, data formats, and authentication.

Key Components of GitHub Actions Workflow

In the official EMPYREAN GitHub repository [4] there are workflow definitions that describe what actions should be taken when a specific event occurs. Workflows are defined in YAML files stored in the ***".github/workflows"*** directory of the project repository.

The main elements of these workflows are:

- **Jobs:** A job is a collection of steps that are executed together on the same runner. Jobs are the main building blocks of a workflow, defining what tasks need to be done. A workflow can have one or more jobs that run either in parallel or sequentially, depending on the established conditions (Figure 5).
- **Steps:** These are individual steps within a job. Each step can execute a bash command or a specific GitHub Action. Steps are executed in order within the context of the job.
- **Actions:** These are reusable commands that can be used in steps to perform specific tasks (e.g., checking out code, running tests, setting up environments). GitHub provides a large number of predefined actions, and users can also create custom actions tailored to their needs.

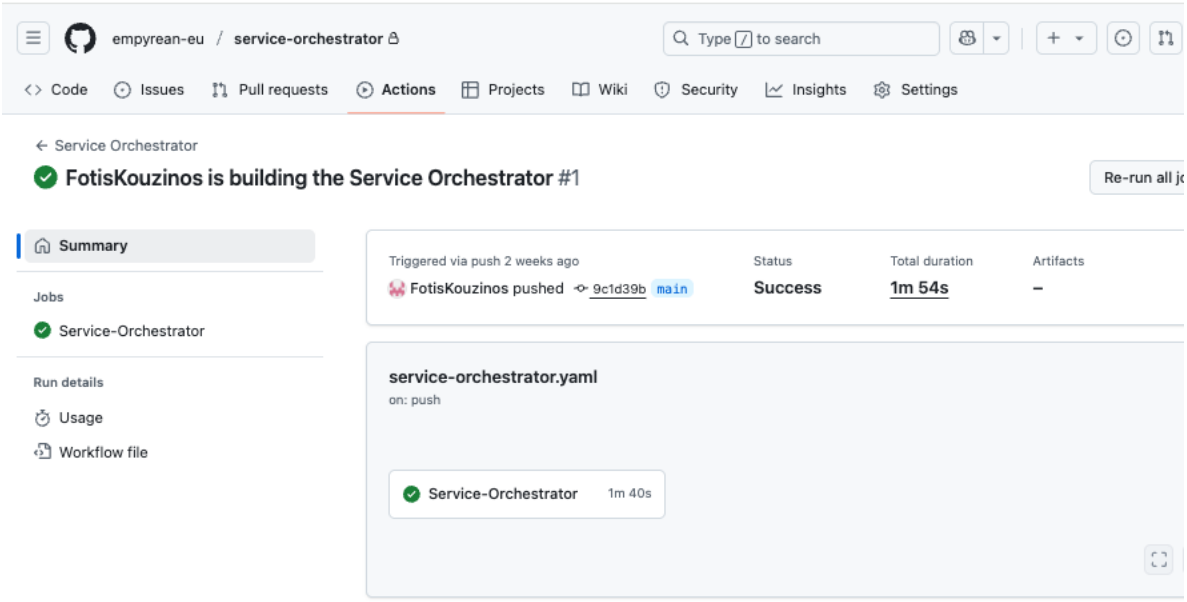


Figure 5: Illustration of the Service Orchestrator job in the GitHub repository.

Secrets in GitHub

Secrets in GitHub are sensitive values, such as API keys or access tokens, that are securely stored and used in GitHub Actions workflows. Secrets are encrypted and only accessible by the repository's workflows. Adding secrets to EMPYREAN Repository is as simple as executing the following steps in the GitHub repository, see Figure 6:

1. In the Settings tab of the repository.
2. In the left menu, select Secrets and variables and then Actions.
3. Click on New organization/repository secret.
4. Enter the name and value of the secret and click on Add secret.

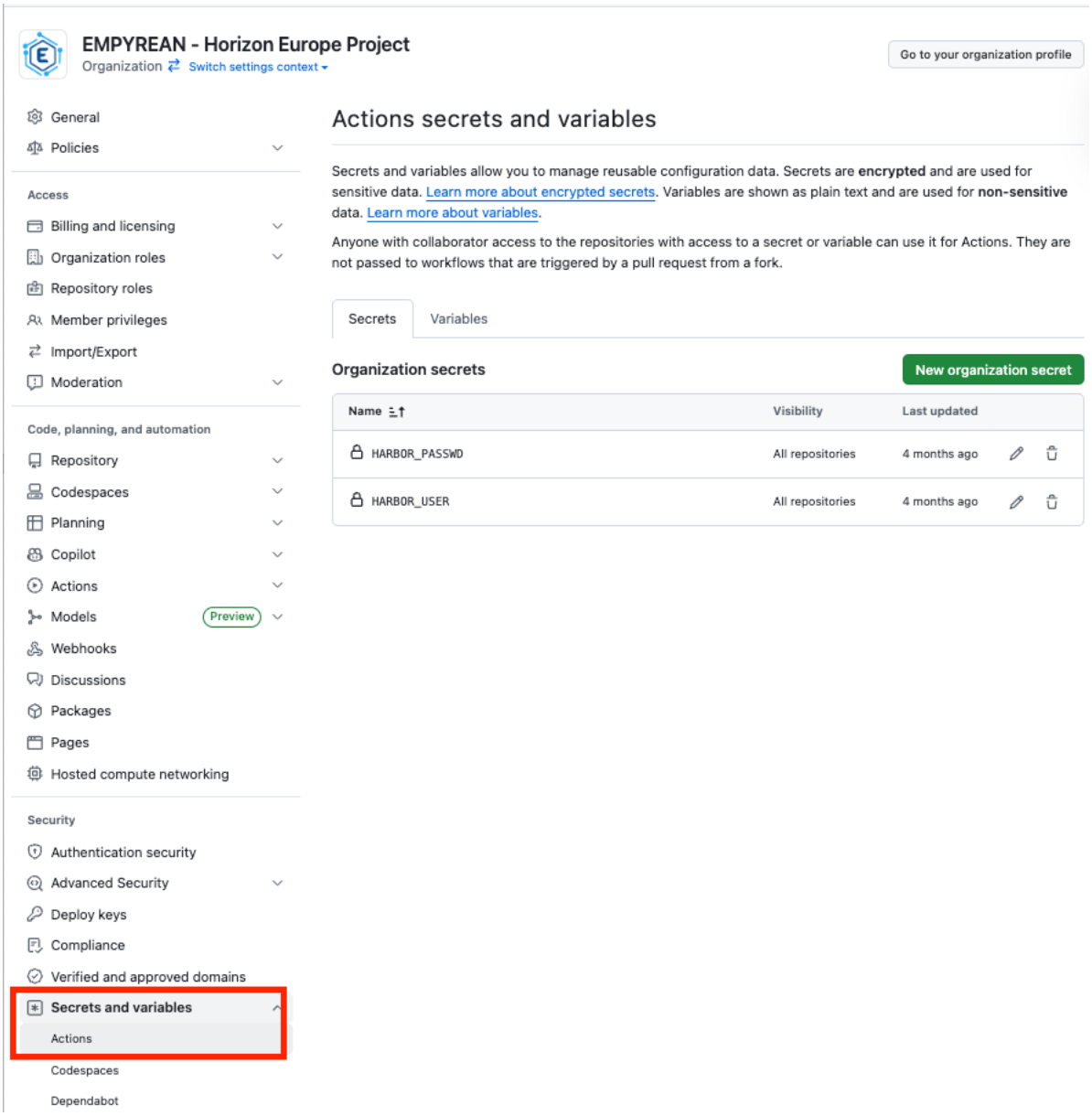


Figure 6: Adding secrets to EMPYREAN Repository.

Using Secrets in a Workflow

To reference a secret in a GitHub Actions workflow, we can use following the syntax `${{secrets.NAME_OF_SECRET}}`.

This allows developers to securely inject sensitive information into workflow steops without exposing them in plain text.

Necessary Secrets for the Build Job of the Workflow

To ensure proper execution of the build job within the EMPYREAN CI/CD pipeline, a series of secrets must be configured in the GitHub repository.

Table 3 presents a detailed overview of the EMPYREAN CI/CD template.

Table 3: EMPYREAN CI/CD template.

EMPYREAN's ci-template.yml

```
name: CD/CI Pipeline
on: push

jobs:
  build:
    runs-on: [base-dind-2204-arm64]

    steps:
      - name: Checkout the repository
        uses: actions/checkout@v3

      - name: SonarQube Analysis
        uses: SonarSource/sonarqube-scan-action@v2
        env:
            SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
            SONAR_HOST_URL: https://sonarqube.k8s-ants.inf.um.es

      - name: Set up Python
        uses: actions/setup-python@v3
        with:
          python-version: 3.9

      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt

      - name: Run unit tests inside runner system
        run: |
          export PYTHONPATH="$PYTHONPATH:$(pwd)"
          pytest tests/

      - name: Build Docker image
        run: docker build -t empyrean-eu/template-image:latest .

      - name: Run tests inside Docker container
        run: |
       docker run --rm empyrean-eu/template-image:latest pytest tests/

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v3

      - name: Login to GitHub Container Registry
       uses: docker/login-action@v3
```

```
45          with:
46          registry: ghcr.io
47          username: ${{ github.actor }}
48          password: ${{ secrets.GITHUB_TOKEN }}
49
50          # Push Docker image to Docker Hub
51          - name: Push Docker image
            run: docker push ghcr.io/empyrean-eu/template-image:latest
```

Workflow name

`name: CD/CI Pipeline`

Workflow Triggers

The workflow is configured to automatically run whenever any changes are made and pushed to the repository.

`on: push`

Jobs

The workflow defines a single job named build, which contains a series of steps to build and test and push the Docker image based on the repository code. The build job runs on a self-hosted runner that we select using the following tags:

```
jobs:
  build:
      runs-on: [base-dind-2204-arm64]
```

Repository Checkout

This step uses the action `actions/checkout@v3` to check out the repository's source code into the workspace. This allows subsequent steps to access the source code needed to build, test and push the Docker image.

```
      steps:
      - name: Checkout the repository
      uses: actions/checkout@v3
```

Code quality analysis

In this step, the action `sonarqube-scan-action@v2` is used in order to launch a code quality analysis in the server `https://sonarqube.k8s-ants.inf.um.es` that will provide information about test coverage, code smells, hotspots, maintainability, reliability, duplications, and security.

```
      - name: SonarQube Analysis
      uses: SonarSource/sonarqube-scan-action@v2
      env:
            SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
            SONAR_HOST_URL: https://sonarqube.k8s-ants.inf.um.es
```

Python Setup

This step uses the action `actions/setup-python@v3` to set up the python version that is needed by the repository code. In this workflow the python version used is the 3.9.

```
- name: Set up Python
uses: actions/setup-python@v3
with:
      python-version: 3.9
```

Dependencies Installation

In this step, the command `pip install` is used to install the dependencies needed by the repository code that are defined in the `requirements.txt` file. Before doing so, it is recommended to `upgrade pip`.

```
- name: Install dependencies
run: |
      python -m pip install --upgrade pip
      pip install -r requirements.txt
```

Unit Tests before Build

This step is meant to run the unit tests inside the runner system before building the image. This shows the test results without having to build the image, which saves the building time in case an output with errors in the pushed code is found. This guarantees a less time-consuming workflow.

```
- name: Run unit tests inside runner system
run: |
      export PYTHONPATH="$PYTHONPATH:$(pwd)"
      pytest tests/
```

Build Docker Image

This step builds the Docker image using the `docker build` command. The image is tagged as `template-image:latest`, which specifies the image name and tag.

```
- name: Build Docker image
run:| docker build -t empyrean-eu/template-image:latest
```

Docker Container Unit Tests

In this step the tests run in the previous step are now run inside the docker container which makes sure the building process worked properly and the image is built as expected.

```
- name: Run tests inside Docker container
run:docker run --rm empyrean-eu/template-image:latest pytest
tests/
```

Docker Buildx Setup

This step sets up Docker Buildx, a tool that enables the building of multi-platform Docker images. The action `docker/setup-buildx-action@v3` is used to prepare the build environment.

```
- name: Set up Docker Buildx
uses: docker/setup-buildx-action@v3
```

Login to GitHub Container Registry

In this step, the login to GitHub Container Registry is performed using credentials stored in repository secrets. The secret GHCR_TOKEN provides the authentication token.

```
- name: Login to GitHub Container Registry
uses: docker/login-action@v3
with:
      registry: ghcr.io
      username: ${{ github.actor }}
      password: ${{ secrets.GITHUB_TOKEN }}
```

Push Docker Image

Publishing the Docker image to the GitHub Container Registry using the following command.

```
- name: Push Docker image
run: docker push ghcr.io/empyrean-eu/template-image:latest
```

4.2 Platform Release Plan

The implementation, integration, validation, and delivery of EMPYREAN developments follow a structured, phased approach (Figure 7) designed to ensure a systematic and iterative progression towards the project's objectives and the release of the final platform. Each phase builds upon the outcomes of the previous one, enabling smooth transitions from requirements analysis and development, to integration and the deployment of a fully functional platform. This approach incorporates continuous feedback and incremental improvements through the development lifecycle.

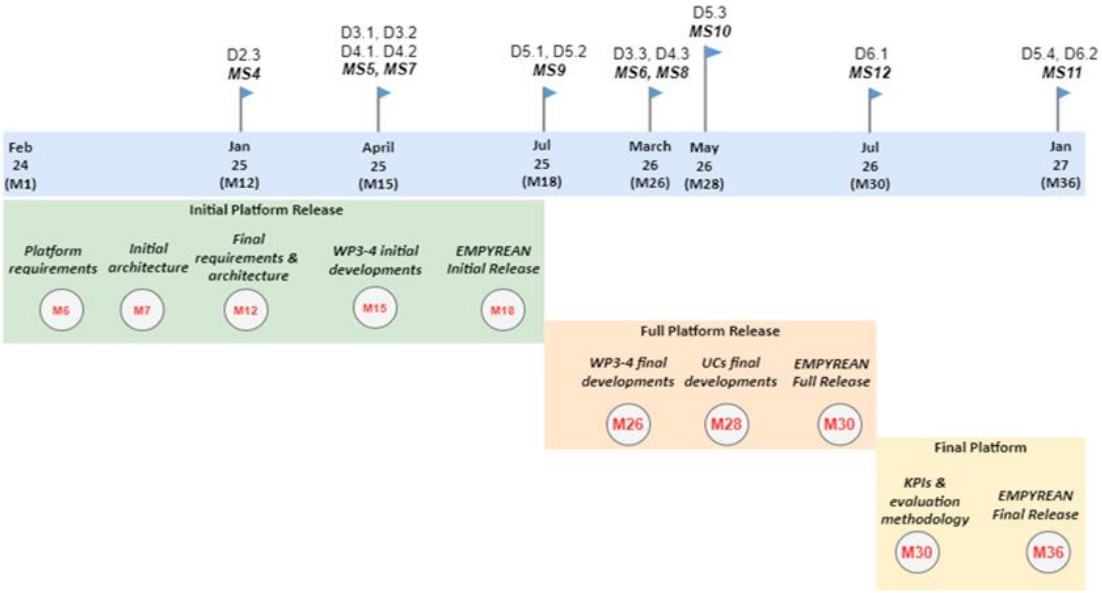


Figure 7: EMPYREAN's platform release roadmap.

In line with this strategy, the EMPYREAN platform is planned to be delivered through three major milestones, presented in Table 4.

Table 4: EMPYREAN Platform's Release Roadmap.

Milestone	Deadline (date)
Initial platform release	M18 (July 2025)
Full platform release	M30 (July 2026)
Final platform release	M36 (Jan 2027)

It is important to emphasize that each phase utilizes an agile approach to Continuous Integration and Continuous Deployment (CI/CD), as discussed in the previous subsections. This method enables us to respond effectively to advancements in the field and emerging technological trends while allowing for continuous improvement based on real-world experimentation.

D5.2 marks the successful release of the initial version of the EMPYREAN platform, developed during the first iteration of the project's implementation (M4–M15). In this version, each component delivers a subset of its intended functionality, along with the core interfaces required for inter-component communication. The initial release serves as a foundational prototype that effectively demonstrates the core capabilities of the EMPYREAN platform while also offering valuable insights and feedback to guide the second development iteration (M18–M36).

Building upon this foundation, the **full platform release** will extend the prototype by incorporating the remaining functionalities not included in the initial version. As the development activities within the technical work packages (WP3 and WP4) conclude by M26, the full release is scheduled for M30. Its objective is to deliver a fully integrated and feature-complete platform, ready to support pilot deployments and experimentation. The **final release**, to be delivered at the end of the project (M36), will focus on refining the platform based on insights gathered from the final evaluation and the demonstration of the project's use cases.

By following this structured and iterative development approach, the EMPYREAN consortium ensures a smooth and effective transition from concept to a fully operational and exploitable platform, in alignment with the project's technical goals and stakeholder needs.

5 EMPYREAN Platform Components

This section describes in detail the platform components, each of which delivers a subset of the targeted features along with the primary interfaces required for inter-component communication.

5.1 Service Layer

5.1.1 Workflow Manager

The Ryax platform is architected as a collection of modular, containerized microservices orchestrated by Kubernetes to enable flexible, scalable, and fault-tolerant deployment of data processing workflows. Based on the initial analysis provided in D4.1 (M15), two critical microservices in this architecture are Studio and Runner. While the Studio microservice enables users to define and configure workflows in a low-code, declarative manner, the Runner microservice is responsible for executing those workflows and managing their runtime behavior. The following sections detail these services from two perspectives: their design and purpose within the platform, and their technical interfaces and integration points within the Ryax system, preparing the way to integrate with other EMPYREAN components.

This section presents the functional roles and architectural responsibilities of the Studio and Runner microservices in the broader context of the Ryax platform. While both services operate independently, they are closely integrated: Studio acts as the control layer that defines workflows, and Runner is the data execution layer that brings them to life. Together, they support Ryax's core objective, enabling users to design and deploy end-to-end data automation pipelines on cloud, edge, and hybrid infrastructures with minimal operational overhead.

Studio

The Studio microservice is the central component responsible for managing the lifecycle and configuration of workflows in the Ryax platform. It empowers users, through both graphical and programmatic interfaces, to create, edit, and deploy complete data processing applications composed of modular building blocks called actions (triggers, processors, publishers). It offers a low-code, declarative workflow design environment, allowing users to define workflows as Directed Acyclic Graphs (DAGs) using an intuitive UI or YAML-based configurations. Each node (action) in the workflow can process data, produce outputs, and pass results downstream in a well-defined data stream model.

Internally, it is implemented as a stateless HTTP REST API using Python. It uses a PostgreSQL database to persist workflow definitions, metadata, and configurations, and applies an ORM (Object-Relational Mapping) layer for maintainable and structured data management. Studio abstracts the complexity of defining data analytics pipelines and container-based deployments. It is tightly integrated with Ryax's internal services such as the Repository,

Action Builder, and Runner, acting as the control plane for pipeline definition and deployment.

Runner

The Runner microservice is the execution engine of the Ryax platform. It is responsible for managing the runtime behavior of workflows defined via the Studio service. Once a workflow is deployed, Runner orchestrates the execution of each action, handles task scheduling and data flow, collects logs, and manages execution metadata.

Runner supports containerized, distributed execution over Kubernetes-managed infrastructures and communicates with deployed actions using gRPC. It integrates with Ryax's MinIO-based Filestore to handle execution inputs and outputs, and uses RabbitMQ as its internal messaging broker for service coordination.

Designed to handle execution at scale, Runner ensures that workflows are executed efficiently, that tasks are retried upon failure, and that results are stored and made available via structured APIs. It also exposes real-time monitoring capabilities to allow users and administrators to track the progress of executions across distributed resources.

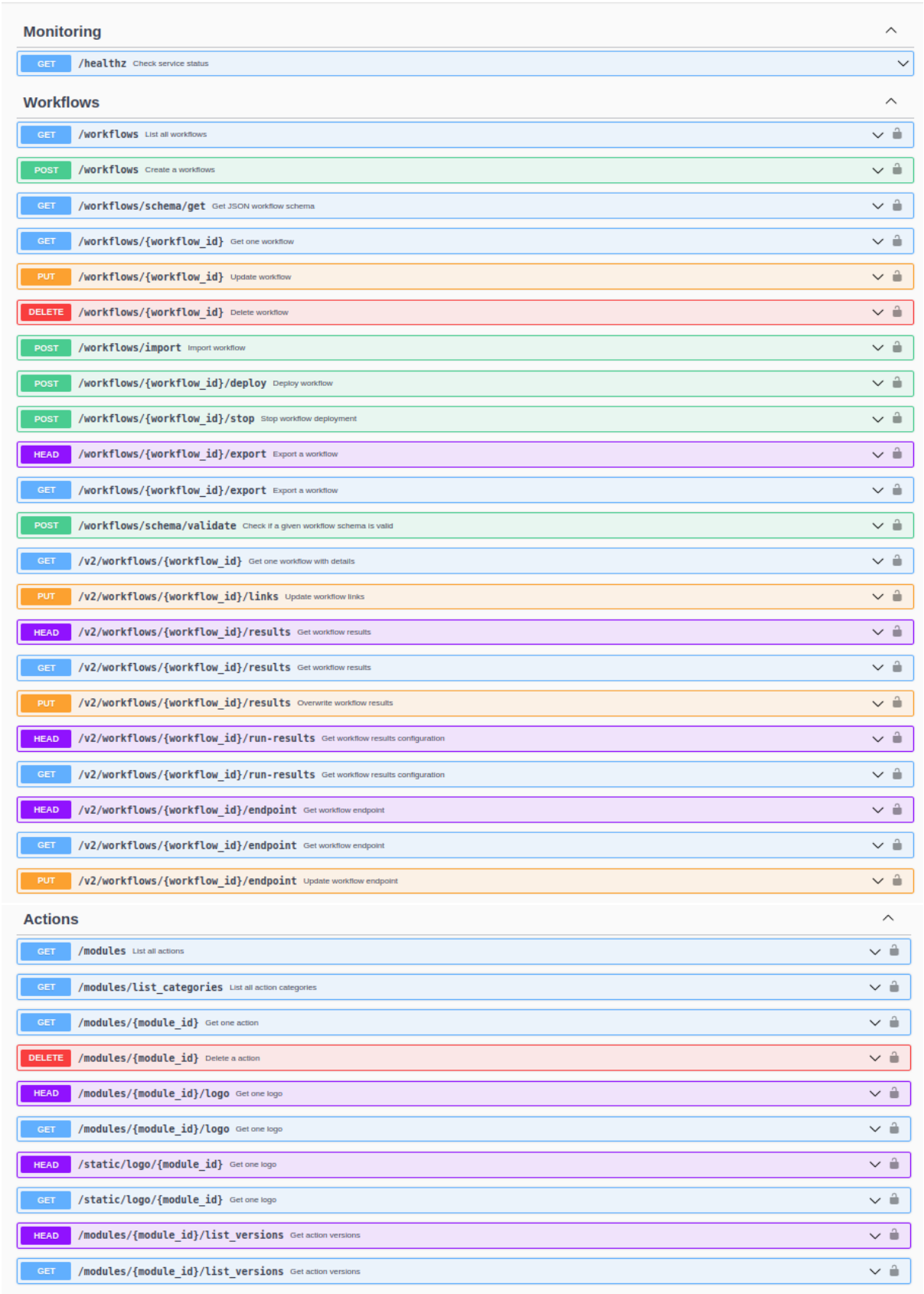
Integration and Interfaces

Studio

The Studio microservice exposes a comprehensive RESTful API (Figure 8) that is consumed by both the Ryax WebUI (Angular-based frontend) and the Ryax CLI (Python-based tool). These interfaces allow users to interact with the platform visually or programmatically.

The API, defined in an OpenAPI (Swagger) specification, includes endpoints for:

- **Workflow lifecycle management:** Create, update, delete, and retrieve workflows.
- **Action and workflow structure management:** Define and link triggers, processors, and publishers as part of a DAG.
- **Input/output configuration:** Specify and manage action parameters and data flow.
- **Deployment and monitoring metadata:** Track deployment state, trigger executions, and view errors.
- **Project-level variables:** Inject reusable parameters and configurations across workflows.


Monitoring
GET /healthz Check service status
Workflows
GET /workflows List all workflows
POST /workflows Create a workflows
GET /workflows/schema/get Get JSON workflow schema
GET /workflows/{workflow_id} Get one workflow
PUT /workflows/{workflow_id} Update workflow
DELETE /workflows/{workflow_id} Delete workflow
POST /workflows/import Import workflow
POST /workflows/{workflow_id}/deploy Deploy workflow
POST /workflows/{workflow_id}/stop Stop workflow deployment
HEAD /workflows/{workflow_id}/export Export a workflow
GET /workflows/{workflow_id}/export Export a workflow
POST /workflows/schema/validate Check if a given workflow schema is valid
GET /v2/workflows/{workflow_id} Get one workflow with details
PUT /v2/workflows/{workflow_id}/links Update workflow links
HEAD /v2/workflows/{workflow_id}/results Get workflow results
GET /v2/workflows/{workflow_id}/results Get workflow results
PUT /v2/workflows/{workflow_id}/results Overwrite workflow results
HEAD /v2/workflows/{workflow_id}/run-results Get workflow results configuration
GET /v2/workflows/{workflow_id}/run-results Get workflow results configuration
HEAD /v2/workflows/{workflow_id}/endpoint Get workflow endpoint
GET /v2/workflows/{workflow_id}/endpoint Get workflow endpoint
PUT /v2/workflows/{workflow_id}/endpoint Update workflow endpoint
Actions
GET /modules List all actions
GET /modules/list_categories List all action categories
GET /modules/{module_id} Get one action
DELETE /modules/{module_id} Delete a action
HEAD /modules/{module_id}/logo Get one logo
GET /modules/{module_id}/logo Get one logo
HEAD /static/logo/{module_id} Get one logo
GET /static/logo/{module_id} Get one logo
HEAD /modules/{module_id}/list_versions Get action versions
GET /modules/{module_id}/list_versions Get action versions

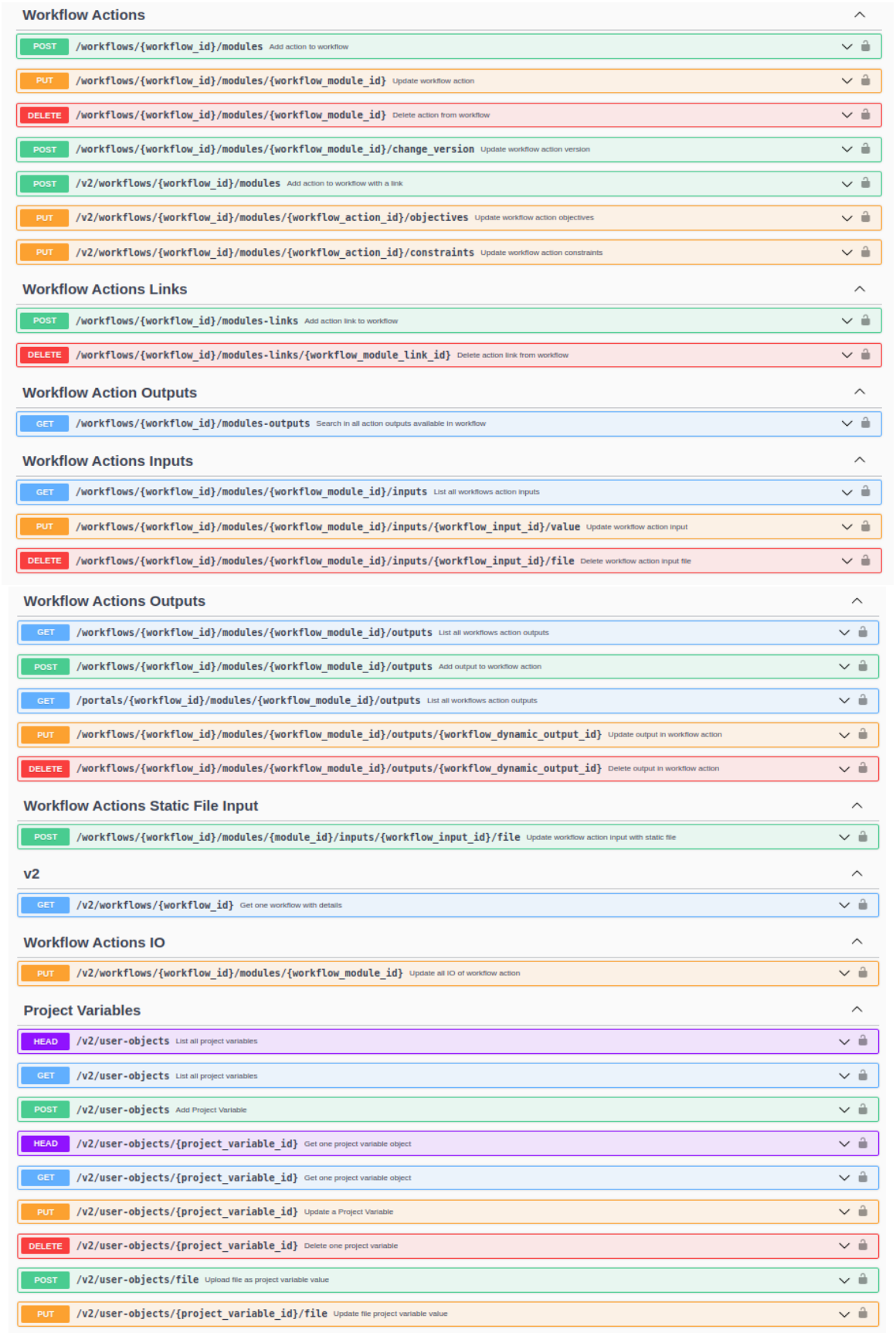


Figure 8: Ryax Studio - RESTful API.

Studio interacts with other internal services such as: (i) *Repository* to import or discover actions from Git, (ii) *Action Builder* to package and prepare actions for deployment, and (iii) *Runner* to trigger executions and manage runtime coordination.

All service-to-service communication occurs over HTTP or asynchronous message passing via RabbitMQ. Authentication and authorization are enforced through integrated user management services.

Runner

The Runner API exposes the interfaces (Figure 9) required to manage workflow execution, track their status, and access runtime results and logs. It complements the Studio API by operationalizing workflow definitions into executable tasks. The API is organized into the following key functional areas:

- **Execution lifecycle management**: Interfaces to start, cancel, and monitor workflow runs, as well as manage run history and execution metadata.
- **Live monitoring and observability**: Endpoints to access execution logs (both batch and streaming), deployment events, and system health checks.
- **Portal access**: Workflow triggering mechanisms designed for end-user or third-party access via portals.
- **Data handling**: APIs to retrieve results, download execution outputs, and access intermediate files stored during execution.
- **Security and access control**: Functionality for API key management and user authentication for protected operations.
- **Infrastructure awareness**: Interfaces for querying available execution sites and collecting accounting data on resource usage.

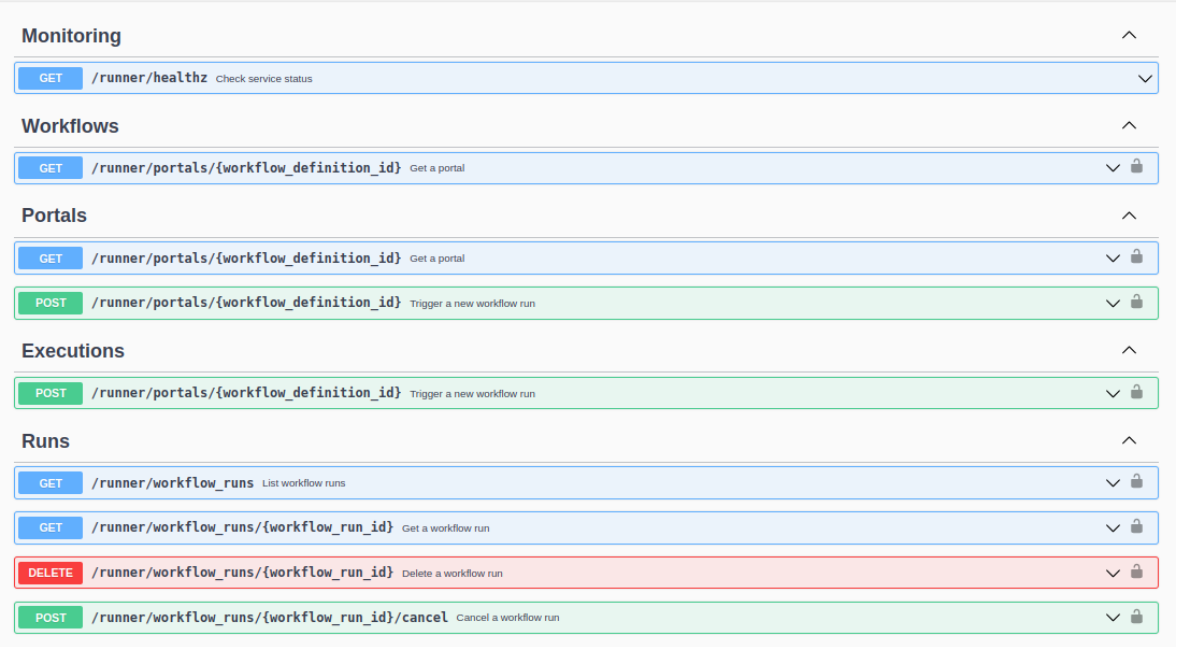


Figure 9: Ryax Runner - RESTful API.

These APIs enable seamless integration of Runner with Ryax's internal monitoring tools, such as Prometheus, Grafana, and Loki, as well as with orchestration systems like Kubernetes and resource-aware scaling mechanisms. Runner also exposes WebSocket endpoints to enable real-time streaming of logs and deployment events, ensuring robust operational visibility.

All communication with Runner is authenticated and scoped to user projects, supporting multi-tenant deployments and secure access control. Internally, it uses gRPC to communicate with action containers and Protobuf-encoded messaging via RabbitMQ for event dispatching across services.

5.1.2 Dataflow Programming

The dataflow programming component is implemented on top of the Eclipse Zenoh network protocol [5]. In the following paragraphs, we provide a high-level understanding of the component's structure, main components, and how they work together to enable distributed computations spanning from cloud to edge devices.

Zenoh-Flow [6] is a Zenoh-based data flow programming framework designed for computations that span from the cloud to the device. The system enables users to define, deploy, and manage distributed dataflows using a declarative approach with dynamic node loading and runtime management capabilities. The system architecture is depicted in Figure 10, it maps directly to the specific Rust crates and modules, providing a clear separation of concerns across the distributed dataflow processing pipeline.

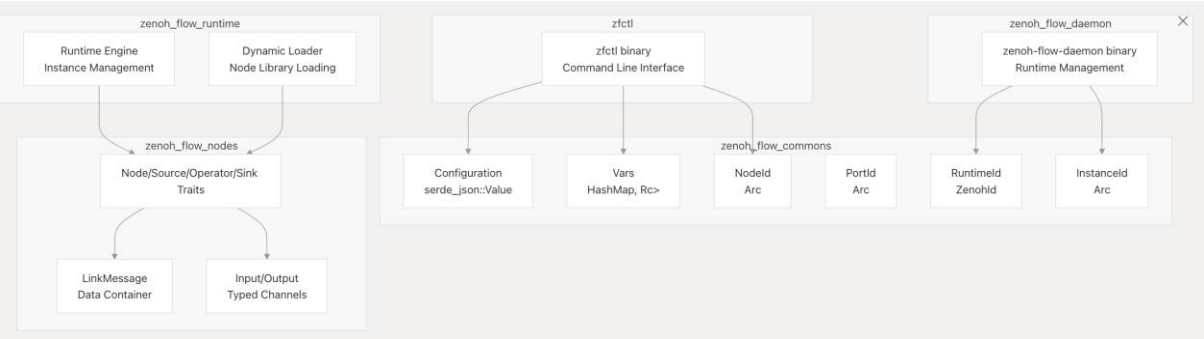


Figure 10: Dataflow framework core components.

The Dataflow component is organized as a Rust workspace containing multiple interconnected crates, each serving specific functionality within the distributed dataflow system.

Integration and interfaces

The command-line interface for Zenoh-Flow is called **zfctl**, which is used to deploy and manage the dataflow instances. It supports the commands to cover the essential workflow from starting daemons to deploying and controlling dataflow instances.

Before starting a new dataflow implementation, the application engineer needs to have the following pre-requisites:

- A working Zenoh network (**zfctl** connects as a peer with multicast scouting by default)
- Compiled node libraries available for the dataflows
- Access to dataflow descriptor files in YAML format

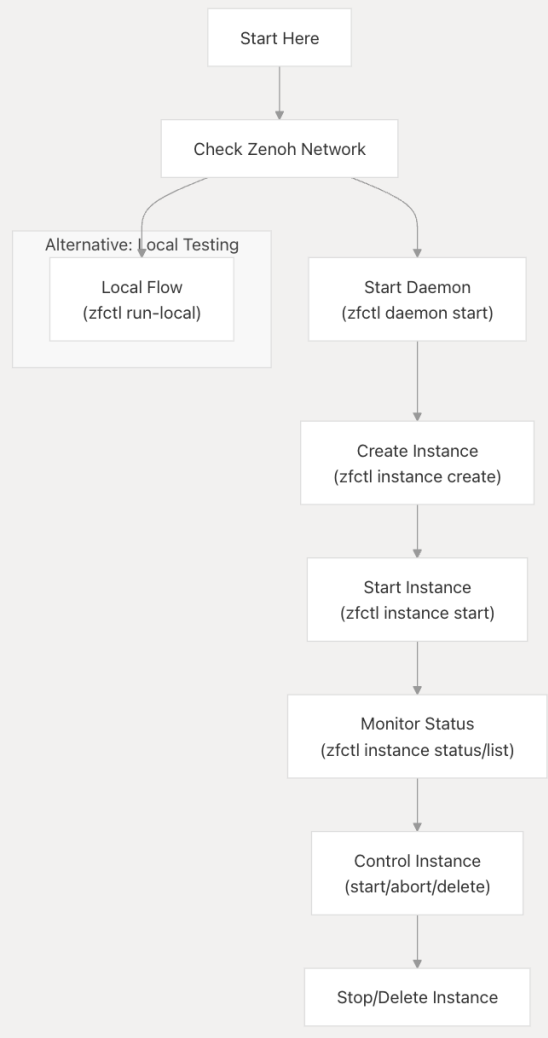


Figure 11: Dataflow getting started workflow.

The typical workflow involves three main phases: daemon management, dataflow deployment, and instance control, see Figure 11. Detailed information can be found in the Online Zenohflow tutorial [5].

5.1.3 Action Packaging

The Action Packaging component of EMPYREAN is realized through two distinct services within the Ryax architecture: the Repository and the Action Builder. These microservices form the backbone of Ryax's action development pipeline, enabling the seamless integration of user-defined logic into workflows. They allow custom actions to be scanned, built, and made available within the Ryax ecosystem.

The Repository service handles source control integration and action registration, while the Action Builder compiles those actions into runnable components using a deterministic build process. This section describes their responsibilities and integration pathways.

This section highlights the functional roles of the Repository and Action Builder microservices. These components enable extensibility in Ryax by providing users with tools to onboard custom logic, manage versioning, and ensure reliable and reproducible builds. This extensibility is central to Ryax's low-code, empowering users to easily incorporate domain-specific code without manual packaging or orchestration.

Repository

The Repository microservice is responsible for integrating external Git repositories into Ryax. Users can register repositories (e.g., on GitHub or GitLab), scan their contents for Ryax-compliant action definitions, and trigger builds of those actions. It serves as the source management and synchronization layer in the Ryax architecture. Internally, Repository stores metadata about imported repositories, actions, and their build status using a PostgreSQL database, abstracted via an ORM. It exposes a RESTful HTTP API and is accessible via both the WebUI and CLI.

Once an action is scanned and a build is triggered, the Repository forwards the build request to the Action Builder service. Upon successful build completion, it registers the built action with the Studio's action store, making it immediately available for use in workflow construction. This microservice ensures a clean separation between source integration, build execution, and workflow design, enabling a robust, scalable, and CI/CD-friendly approach to managing custom actions.

Action Builder

The Action Builder microservice is responsible for compiling and packaging user-defined Ryax actions received from the Repository service. It enforces build isolation, dependency resolution, and reproducibility using the Nix package manager, a purely functional system designed to create deterministic builds. Build requests are processed sequentially and synchronously, with each build occupying the entire builder service until completion. This design simplifies resource usage and debugging, although it limits concurrency to one build at a time per builder instance.

The Action Builder performs the following steps for each build:

1. Validates the action's structure and metadata.
2. Resolves and installs required dependencies using Nix.
3. Packages the action into a Docker-compatible runtime unit.
4. Notifies the Repository service upon completion or failure.

As a purely backend component, Action Builder does not interact directly with the user-facing WebUI or CLI but acts as a worker that listens for instructions from Repository. It communicates results via API callbacks and logging channels.

Integration and Interfaces

This section explains how the Repository and Action Builder services integrate with broader Ryax system. Exposed via REST APIs and documented with OpenAPI, they support seamless integration with automation tools, CI/CD systems, and Ryax's internal services. They are accessible through both CLI and WebUI for streamlined development workflows.

Repository

The Repository API (Figure 12) provides endpoints that allow users to:

- **Manage Git repositories**: Add, update, delete, and retrieve source repositories (*/sources, /v2/sources*)
- **Scan repositories**: Detect Ryax-compatible action definitions from Git-based sources (*/v2/sources/{id}/scan*)
- **Trigger action builds**: Initiate builds for all or individual actions in a repository (*/v2/sources/{id}/build, /modules/{id}/build*)
- **Manage repository actions**: List or delete imported actions (*/modules, /modules/{id}*)
- Handle build operations: Cancel ongoing builds either per action or per repository (*/v2/actions/{id}/cancel_build, /v2/sources/{id}/cancel_all_builds*)
- **Monitor health**: Check service status (*/healthz*)

The WebUI uses the API to enable developers to link external code and build Ryax actions directly from the browser. Similarly, CLI-based users can automate action imports and updates within CI/CD pipelines. The Studio service is notified upon successful builds, allowing actions to be added to the platform's shared action catalog.

Internally, the Repository communicates with the Action Builder through REST-based build request calls and uses PostgreSQL to track repository state, action metadata, and build history.

Action Builder

Although it does not expose user-facing endpoints directly, the Action Builder operates via a dedicated internal interface used exclusively by the Repository service. It performs one build job at a time, processing incoming tasks in FIFO order.

Key integration characteristics include:

- **Tightly coupled build queue**: Build jobs are dispatched directly from Repository as synchronous API calls.
- **Nix-based reproducibility**: Ensures that all dependencies for custom actions are resolved in a clean, sandboxed environment.
- **Result reporting**: Status updates (success or failure) are relayed back to the Repository, which then triggers any required downstream actions, such as Studio registration or error notification.

Because of its dependency on Nix and synchronous job model, Action Builder is typically deployed with constrained concurrency, but it can be horizontally scaled with job distribution logic in the Repository for high-volume environments.

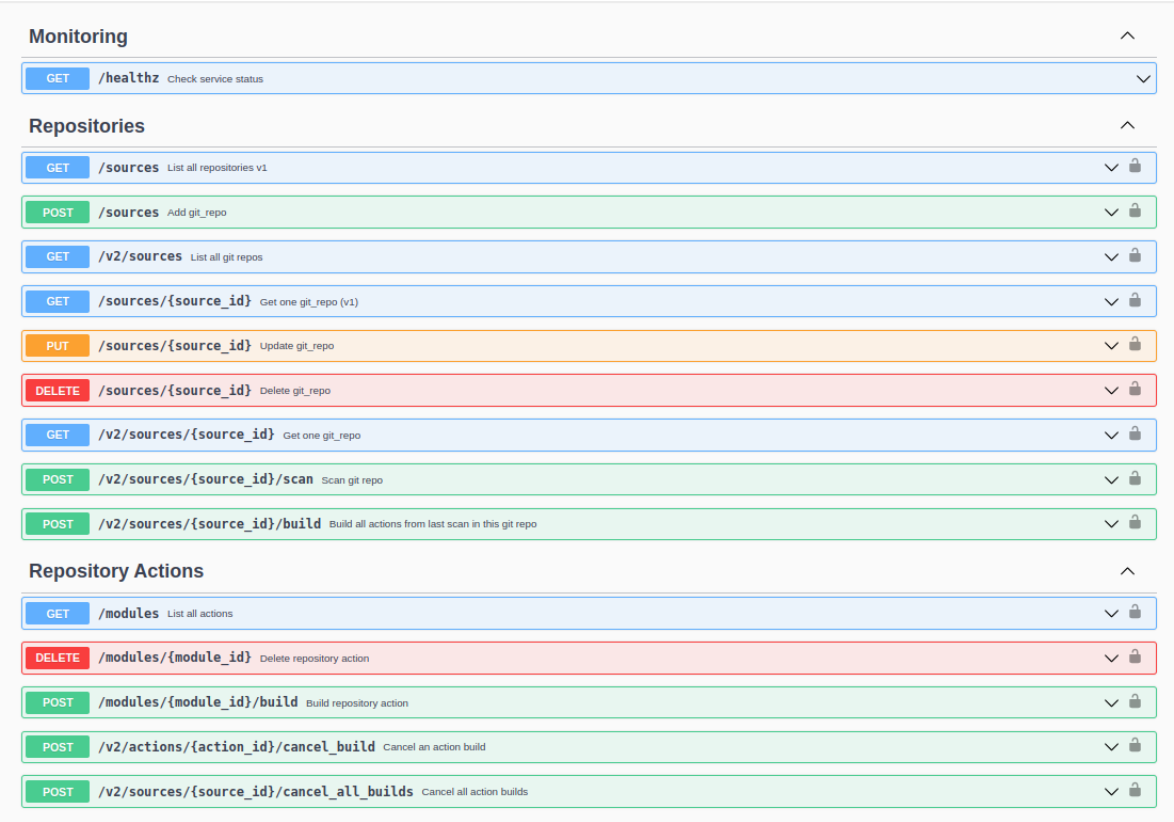


Figure 12: Action Packaging – Repository microservice RESTful API.

5.1.4 Unikernel Building

Traditional cloud-native deployments use POSIX-like containers, which can be inefficient for lightweight tasks such as network services or serverless functions. The Unikernels Builder (Bunny) enables deployment of unikernel images, single-purpose, bootable VMs without extra dependencies, fulfilling EMPYREAN's need for secure, scalable, lightweight applications across IoT-edge-cloud environments.

Bunny automates building, packaging (OCI-compatible), publishing, and deploying unikernel apps, and integrates with container registries and Kubernetes via urunc, allowing unikernels to run in standard cloud-native deployment workflows.

With Bunny, EMPYREAN achieves flexible deployments that minimize memory footprint, boot time, and attack surface, crucial for edge and embedded systems in sensitive contexts.

Integration and interfaces

To align with current cloud-native standards, Bunny supports OCI-compliant image packaging, ensuring compatibility with popular container registries and orchestration platforms.

Applications intended for unikernel deployment can now leverage a familiar interface akin to Dockerfile, allowing developers to define build steps, dependencies, and runtime configuration in a declarative manner. This streamlined workflow lowers the barrier for onboarding and integration, as existing containerization knowledge translates directly to the unikernel packaging process.

Bunnyfiles offer a Dockerfile-like experience tailored specifically for unikernel builds. Table 5 shows the currently supported options for a bunnyfile. Building such an OCI image is done via the generic Docker interface (*`docker build -f Bunnyfile -t TAG ./context`*).

Table 5: Unikernel building - Bunnyfile syntax

```
#syntax=harbor.nbfc.io/nubificus/bunny:latest   # [1] Set bunnyfile syntax for automatic recognition from buildkit.
version: v0.1                                   # [2] Bunnyfile version.

platforms:                                      # [3] The target platform for building/packaging.
  framework: unikraft                           # [3a] The unikernel framework.
  version: v0.15.0                              # [3b] The version of the unikernel framework.
  monitor: qemu                                 # [3c] The hypervisor/VMM or any other kind of monitor.
  architecture: x86                             # [3d] The target architecture.

rootfs:                                         # [4] (Optional) Specifies the rootfs of the unikernel.
  from: local                                   # [4a] (Optional) The source or base of the rootfs.
  path: initrd                                    # [4b] (Required if from is not scratch) The path in the source,
where the prebuilt rootfs file resides.
  type: initrd                                  # [4c] (optional) The type of rootfs (e.g. initrd, raw, block)
  include:                                      # [4d] (Optional) A list of local files to include in the rootfs
    - src:dst

kernel:                                         # [5] Specify a prebuilt kernel to use
  from: local                                   # [5a] Specify the source of a prebuilt kernel.
  path: local                                   # [5b] The path where the kernel image resides.

cmdline: hello                                  # [6] The cmdline of the app.
```

5.2 Association Management Layer

5.2.1 EMPYREAN Aggregator

The EMPYREAN Aggregator forms the management fabric of the Association-based continuum, enabling seamless coordination and control of distributed resources and services. The Aggregator oversees resource provisioning, workload scheduling, and interconnection across Associations. Aggregators interact not only with each other but also with underlying edge infrastructures and multi-cloud environments, ensuring cohesive, scalable, and adaptive management. This structure enables localized decision-making while maintaining global coordination, promoting efficiency and autonomy.

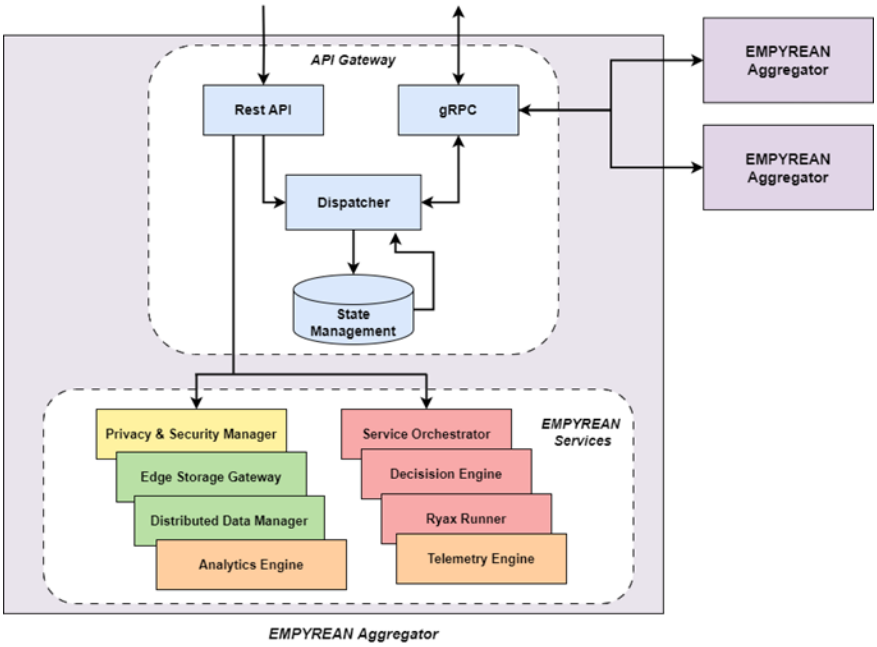


Figure 13: EMPYREAN Aggregator architecture.

The Aggregator consolidates multiple EMPYREAN services and components (Figure 13). It serves as an abstraction point between Service and Multi-Cluster Orchestration layers, ensuring composability and interoperability across the continuum. Further details on the design and initial design of this component can be found in D4.2 (M15) [7].

Integration and interfaces

The EMPYREAN Aggregator is a cloud-native application developed in Python. Its core services are containerized and managed using Kubernetes for orchestration. To enable efficient development and integration, the Aggregator is fully integrated into the EMPYREAN CI/CD pipeline. After the testing phase, container images of the related services are automatically built and made available in the official EMPYREAN image repository. Additionally, all necessary Kubernetes deployment descriptors, such as ConfigMaps, Deployments, and Services, are provided to facilitate automated deployment and lifecycle management within the EMPYREAN integration infrastructure, see Figure 14.

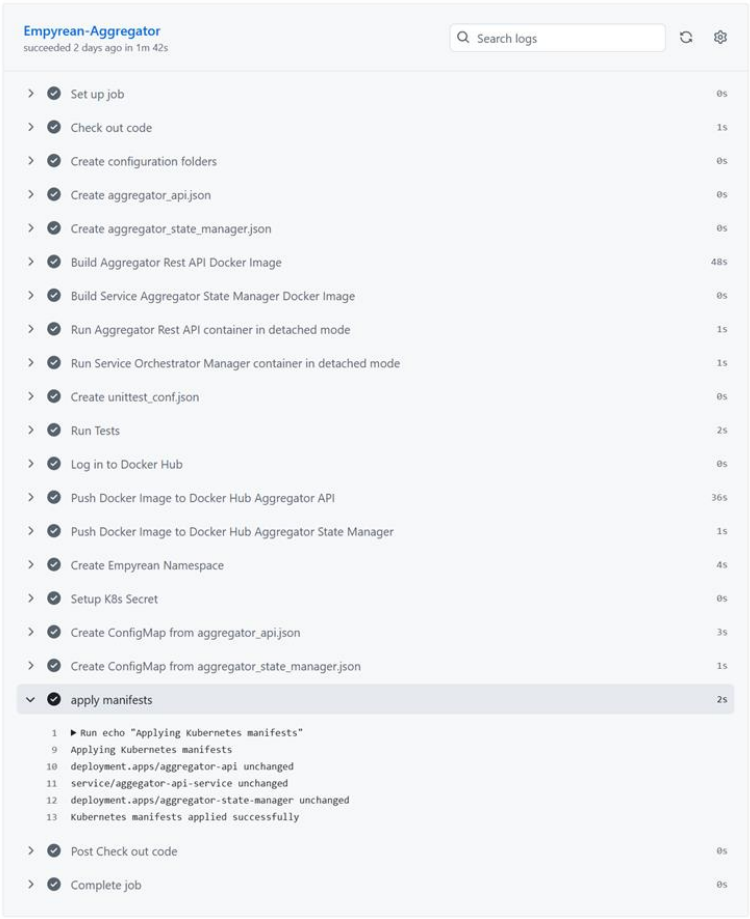
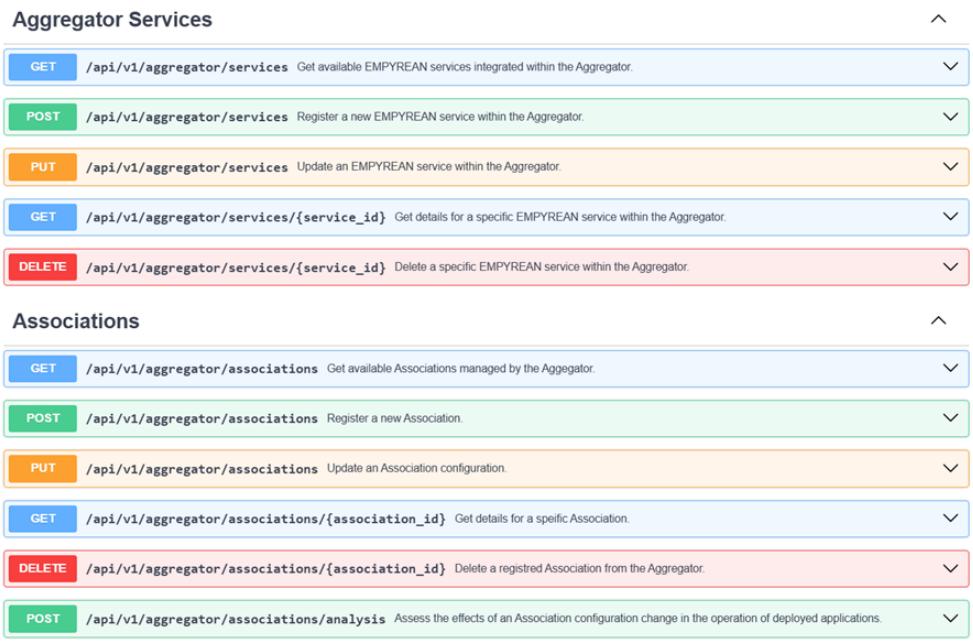


Figure 14: EMPYREAN Aggregator into CI/CD pipeline and deployment in two K8s clusters.

The API Gateway provides two northbound interfaces (NBIs): (i) a REST interface (Figure 15) exposed to core EMPYREAN orchestration and management services, such as EMPYREAN Registry and Workflow Manager, and (ii) a gRPC interface exposed to the rest of the EMPYREAN Aggregators.



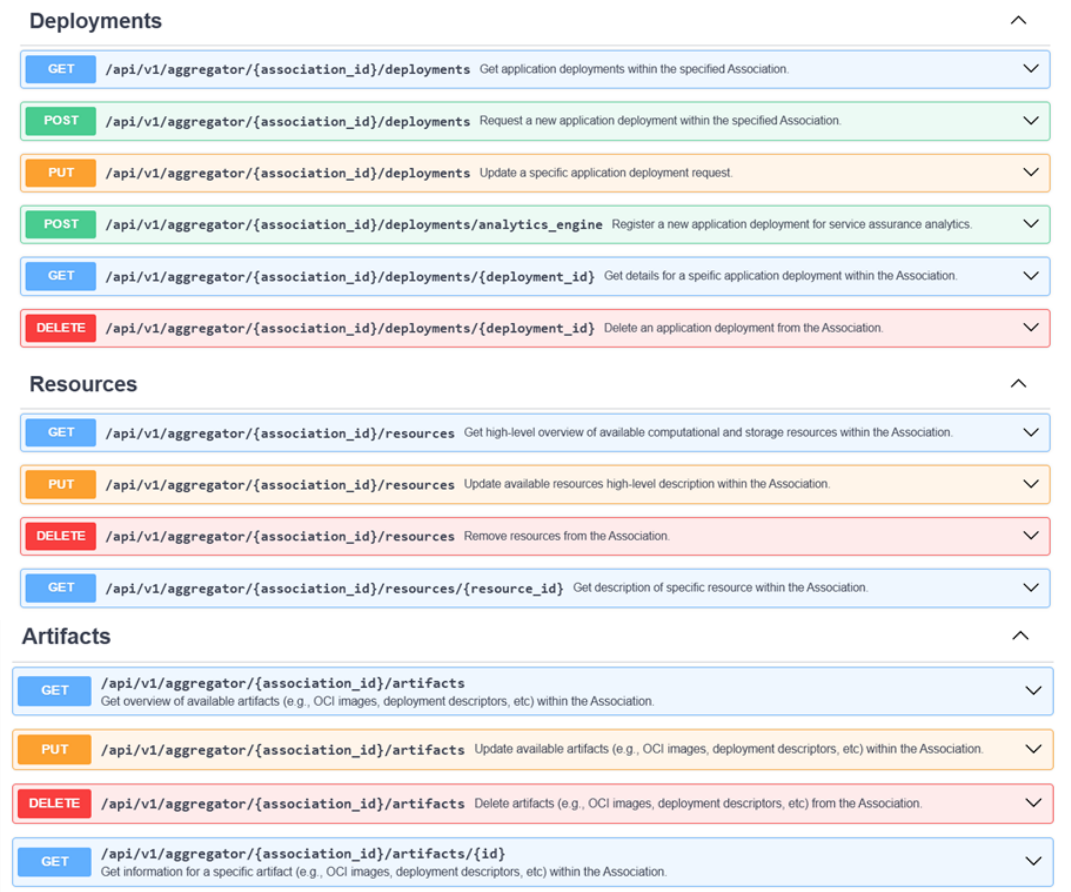


Figure 15: EMPYREAN Aggregator – API Gateway RESTful API.

We provide representative interactions among the EMPYREAN Aggregator and other EMPYREAN platform components to facilitate core operational flows. These examples highlight both REST and gRPC-based communications, covering actions such as onboarding resources, retrieving information, and supporting cross-Aggregator coordination.

Association with the Aggregator assignment for management

When a new Association is defined, the API Gateway of the EMPYREAN Registry notifies the corresponding Aggregator to take control for managing and coordinating the Association. This is achieved by invoking the following POST method exposed by the Aggregator's RESTful API.

```
POST /api/v1/aggregator/associations
```

```
{"uuid": "ebe12f54-a9cb-476a-b930-4befb0236fce", "name": "ICCS Default Association",
"labels": ["platform-arch:amd64", "platform-arch:arm64"], "aggregator_uuid":
"a2b66fa3-3f13-416b-8644-56328e0de4ba", "owner_uuid":"71c1642b-69c9-4e25-abcc-
dda116e8becd", "policy_uuid":"5c0eed16-31a8-467d-83db-ef1010466755", "description":""}
```

Figure 16 presents the log messages from the EMPYREAN Aggregator.

Figure 16: Log messages from the EMPYREAN Aggregator.

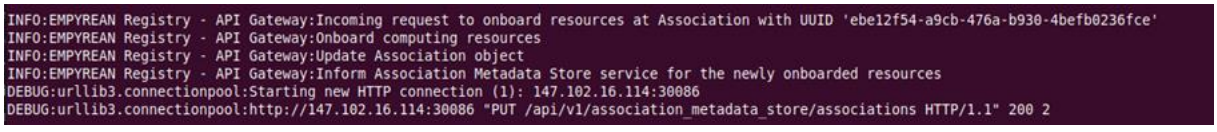
Onboarding computational resources

During the onboarding process, the resource owner provides the EMPYREAN Aggregator, responsible for managing the targeted Association, with a description of the resource's characteristics and applicable policies. The Aggregator then performs the overall process, orchestrating the necessary interactions with various EMPYREAN services, including the EMPYREAN Registry. This is achieved by invoking a PUT request to the Aggregator, where the request body specifies the resource type (via the *resource_type* field) and resources identity parameters (via the *resources* field). Below is an example illustrating the onboarding of three worker nodes belonging to two different clusters

```
PUT /api/v1/aggregator/ebe12f54-a9cb-476a-b930-4befb0236fce/resources

{ "resource_type": "computing_resources","resources": [{"kind":
"worker_node","cluster_uuid":"7628b895-3a91-4f0c-b0b7-
033eab309891","machine_id":"02e257de949c4486b85ba436ec983663","name":"wn1-
paros","policy_uuid":"dd9ceff4-667e-4704-bb06-88e3272cbb27","owner_uuid":"8a99c456-
ff60-44aa-8a79-ca58fe9f6b2d"},{"kind":"worker_node","cluster_uuid":"7628b895-3a91-
4f0c-b0b7-033eab309891","machine_id":"dc156c5469db44c0be8121e8b94e31f6","name":"wn2-
serifos","policy_uuid":"dd9ceff4-667e-4704-bb06-88e3272cbb27","owner_uuid":"8a99c456-
ff60-44aa-8a79-ca58fe9f6b2d"},{"kind":"worker_node","cluster_uuid": "2b05cfdf-679f-
45f1-95f8-a334ec87faaf","machine_id": "f51aa228c77741e797f8d8fe6ac29efe","name": "wn1-
ios","policy_uuid": "dd9ceff4-667e-4704-bb06-88e3272cbb27","owner_uuid":"8a99c456-
ff60-44aa-8a79-ca58fe9f6b2d"}]}}
```

Log messages from the EMPYREAN Registry and graphical representation of the updated Association information within the Association Metadata Store service.



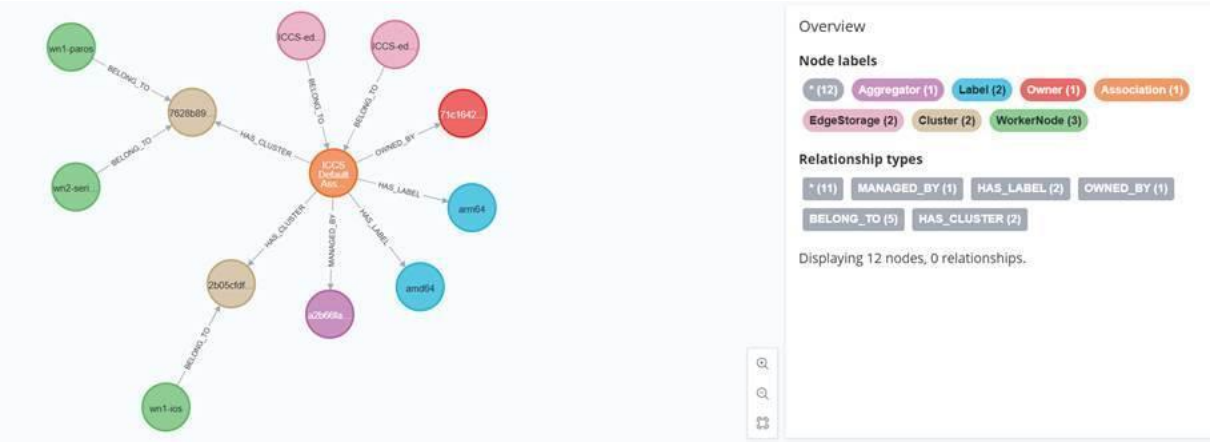


Figure 17: Graphical representation of the updated Association information within the Association Metadata Store service.

Onboarding edge storage resources

Edge storage devices are onboarded in a similar manner to Associations. The Storage Service makes a PUT request to the corresponding Aggregator, supplying the required configuration details. Upon receiving this request, the Aggregator updates its internal state accordingly and notifies the EMPYREAN Registry to reflect the updated capabilities of the Association. The following request onboards two edge storage devices within Association with UUID *"ebe12f54-a9cb-476a-b930-4befb0236fce"*.

```
PUT /api/v1/aggregator/ebe12f54-a9cb-476a-b930-4befb0236fce/resources
```


```
{"resource_type": "storage_resources", "resources": [{"edge_storage_uuid":"7381fd0a-ce7e-401e-86b0-c163c1f06c93", "name": "ICCS-edge-storage-1", "lat":37.983810, "lng":23.727539}, {"edge_storage_uuid":"263020f1-5f19-44a0-a358-2edb905798aa", "name": "ICCS-edge-storage-2", "lat":37.983810, "lng":23.727539}]}
```


Log messages from the EMPYREAN Aggregator.

```
INFO:EMPYREAN Aggregator - API Gateway: 147.102.22.141:54774 - "PUT /api/v1/aggregator/ebe12f54-a9cb-476a-b930-4befb0236fce/resources HTTP/1.1" 200 OK
INFO:EMPYREAN Aggregator - API Gateway:Update available resources resources in Association with UUID 'ebe12f54-a9cb-476a-b930-4befb0236fce'
DEBUG:EMPYREAN Aggregator - API Gateway:{'resource_type': 'storage_resources', 'resources': [{'edge_storage_uuid': '7381fd0a-ce7e-401e-86b0-c163c1f06c93', '
name': 'ICCS-edge-storage-1', 'lat': 37.98381, 'lng': 23.727539}, {'edge_storage_uuid': '263020f1-5f19-44a0-a358-2edb905798aa', 'name': 'ICCS-edge-storage-2
', 'lat': 37.98381, 'lng': 23.727539}]}
INFO:EMPYREAN.Aggregator.Dispatcher:Handle request for onboarding resources to Association with UUID 'ebe12f54-a9cb-476a-b930-4befb0236fce'
INFO:EMPYREAN.Aggregator.Dispatcher:Onboard storage resources
DEBUG:EMPYREAN.Aggregator.Dispatcher:Detect '2' new storage resources for onboarding
DEBUG:EMPYREAN.Aggregator.Dispatcher:[{'edge_storage_uuid': '7381fd0a-ce7e-401e-86b0-c163c1f06c93', 'name': 'ICCS-edge-storage-1', 'lat': 37.98381, 'lng': 2
3.727539}, {'edge_storage_uuid': '263020f1-5f19-44a0-a358-2edb905798aa', 'name': 'ICCS-edge-storage-2', 'lat': 37.98381, 'lng': 23.727539}]
INFO:EMPYREAN.Aggregator.Dispatcher:Update Aggregator's internal information for available resources within Association with UUID: 'ebe12f54-a9cb-476a-b930-
4befb0236fce'
INFO:EMPYREAN.Aggregator.Dispatcher:Inform Aggregator's State Management service for the newly onboarded resources
INFO:EMPYREAN.Aggregator.Dispatcher:Inform EMPYREAN Registry for the newly onboarded resources
DEBUG:urllib3.connectionpool:Starting new HTTP connection (1): 147.102.16.114:30150
DEBUG:urllib3.connectionpool:http://147.102.16.114:30150 "PUT /api/v1/registry/associations/ebe12f54-a9cb-476a-b930-4befb0236fce/onboard HTTP/1.1" 201 2
```

Log messages from the EMPYREAN Registry and graphical representation of the updated Association information within the Association Metadata Store service.

```
INFO:EMPYREAN Registry - API Gateway:Incoming request to onboard resources at Association with UUID 'ebe12f54-a9cb-476a-b930-4befb0236fce'
INFO:EMPYREAN Registry - API Gateway:Onboard storage resources
DEBUG:EMPYREAN Registry - API Gateway:[{'edge_storage_uuid': '7381fd0a-ce7e-401e-86b0-c163c1f06c93', 'name': 'ICCS-edge-storage-1', 'lat': 37.98381, 'lng': 2
3.727539}, {'edge_storage_uuid': '263020f1-5f19-44a0-a358-2edb905798aa', 'name': 'ICCS-edge-storage-2', 'lat': 37.98381, 'lng': 23.727539}]
INFO:EMPYREAN Registry - API Gateway:Update Association object
INFO:EMPYREAN Registry - API Gateway:Inform Association Metadata Store service for the newly onboarded resources
DEBUG:urllib3.connectionpool:Starting new HTTP connection (1): 147.102.16.114:30086
DEBUG:urllib3.connectionpool:http://147.102.16.114:30086 "PUT /api/v1/association_metadata_store/associations HTTP/1.1" 200 2
```

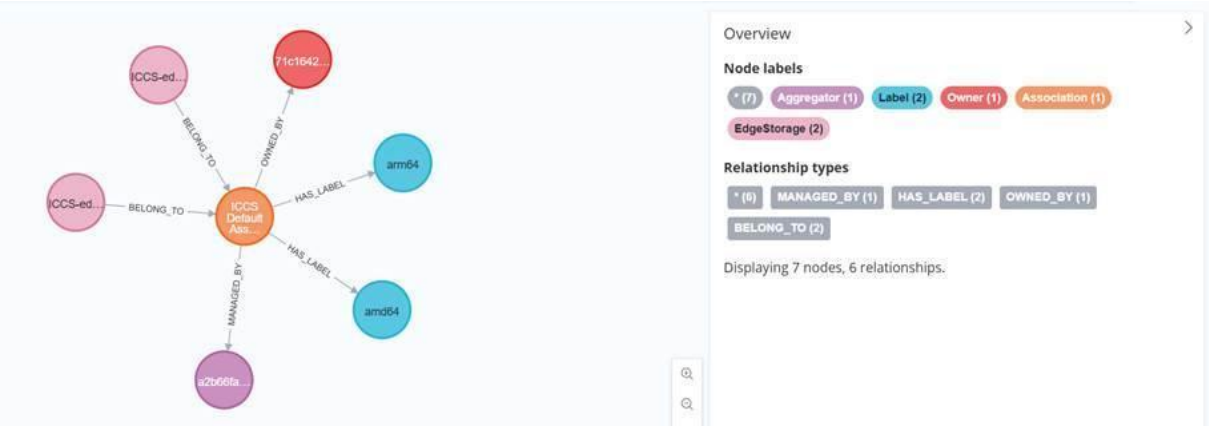


Figure 18: Graphical representation of the updated Association information within the Association Metadata Store service.

Association details

The Aggregator exposes a RESTful endpoint that allows the retrieval of detailed information about a managed Association. The following GET method returns details for the Association with UUID *"ebe12f54-a9cb-476a-b930-4befb0236fce"*, including information on the onboarded computing and storage resources under its management.

```
GET /api/v1/aggregator/associations/ebe12f54-a9cb-476a-b930-4befb0236fce

{"uuid":"ebe12f54-a9cb-476a-b930-4befb0236fce","name":"ICCS Default Association",
"labels":["platform-arch:amd64","platform-arch:arm64"],"aggregator_uuid":"a2b66fa3-
3f13-416b-8644-56328e0de4ba","owner_uuid":"71c1642b-69c9-4e25-abcc-
dda116e8becd","policy_uuid":"5c0eed16-31a8-467d-83db-
ef1010466755","description":"","status":"READY","clusters":["7628b895-3a91-4f0c-b0b7-
033eab309891"],"storage_resources":[{"edge_storage_uuid":"7381fd0a-ce7e-401e-86b0-
c163c1f06c93","name":"ICCS-edge-storage-1", "lat": 37.98381,"lng":
23.727539},{"edge_storage_uuid": "263020f1-5f19-44a0-a358-2edb905798aa","name": "ICCS-
edge-storage-2","lat": 37.98381,"lng":
23.727539}],"computing_resources":[{"cluster_uuid": "7628b895-3a91-4f0c-b0b7-
033eab309891", "machine_id":"02e257de949c4486b85ba436ec983663","name":"wn1-
paros"},{"cluster_uuid": "7628b895-3a91-4f0c-b0b7-
033eab309891","machine_id":"dc156c5469db44c0be8121e8b94e31f6","name":"wn2-
serifos"}],"schedulable": true, "created_at": 1750087208,"updated_at": 1750087217}
```

5.2.2 EMPYREAN Registry

The EMPYREAN Registry plays a central role in enabling coordination, management, and governance across the EMPYREAN platform's complex, distributed environment. Acting as a unified entry point, it supports both core platform services and third-party entities by enabling the discovery, cataloguing, and advertising of Associations, services, and infrastructure components within the Association-based continuum.

Key responsibilities of the Registry include:

- Facilitating the registration and lifecycle of IoT devices, edge, and cloud resources within Associations.
- Maintaining up-to-date information on available Associations, their associated services, and the mapping of resources to each Association.

Capturing and managing the relationships between users and Associations, ensuring consistent and policy-compliant platform behaviour.

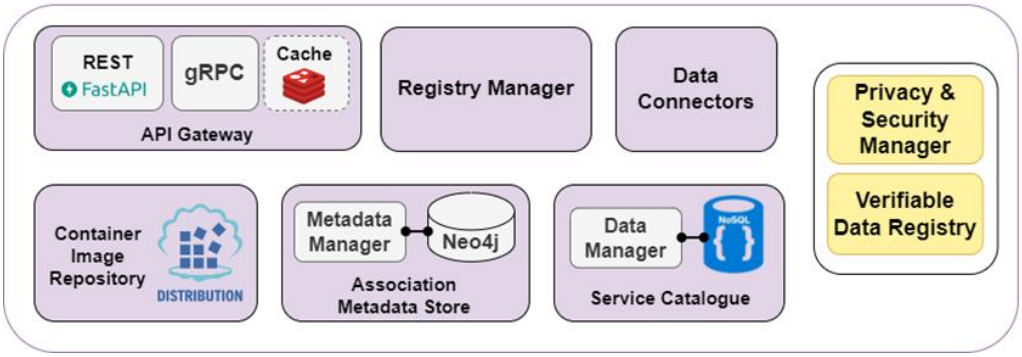


Figure 19: EMPYREAN Registry architecture.

Its architecture, see Figure 19, consists of seven core services, each exposing well-defined interfaces to support interoperability, extensibility, and robust communication across platform components. Further details on the design and initial design of this component can be found in Deliverable D4.2 (M15) [7].

Integration and interfaces

The work during the initial phase covers the implementation of the API Gateway, Association Metadata Store, Service Catalogue, and Container Image Repository components. The Container Image Repository is based on the CNCF Distribution Registry[2], 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. The other components are implemented in Python and are containerized to ensure portability and isolation. The modular architecture allows individual services to evolve independently while interacting through internal REST APIs, ensuring loose coupling and ease of maintenance.

To streamline development, testing, and deployment workflows, the Registry is integrated into the EMPYREAN CI/CD pipeline (Figure 20).

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

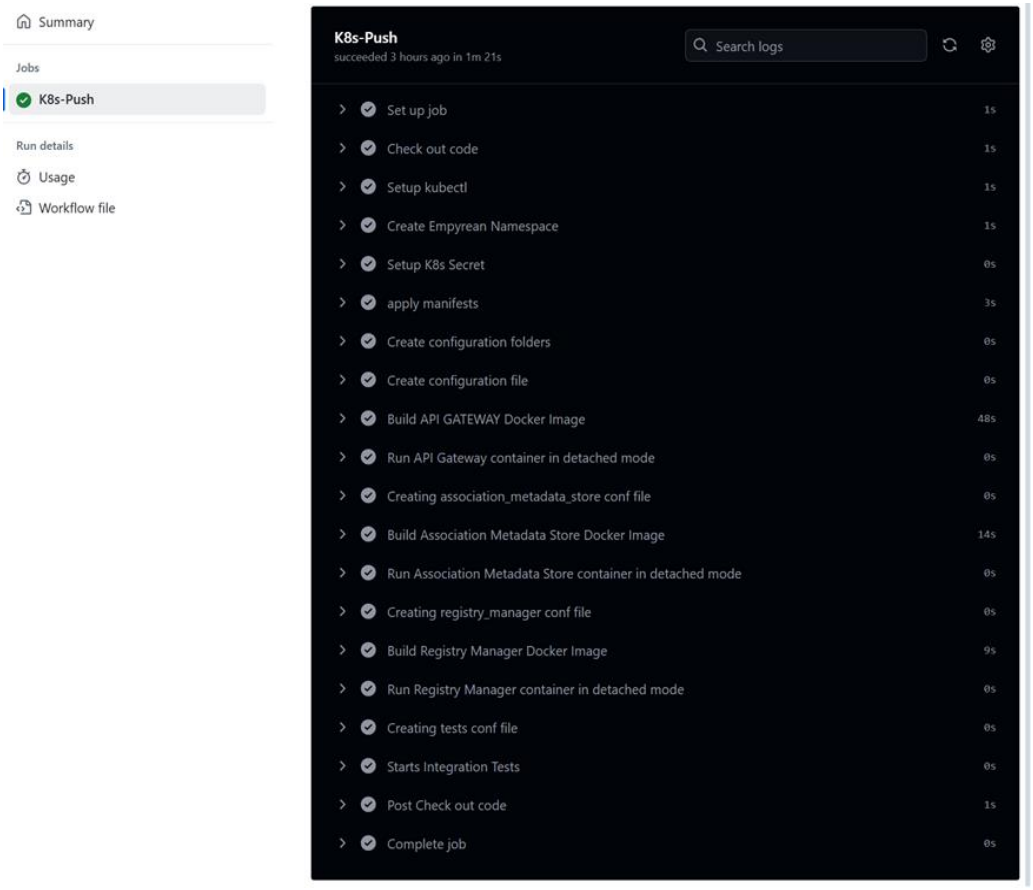


Figure 20: EMPYREAN Registry into EMPYREAN CI/CD pipeline.

Container images for the developed Registry components are automatically built and pushed to the EMPYREAN image repository. Corresponding Kubernetes YAML descriptors are maintained to support automated deployment in test, staging, and production environments (Figure 21).

```
(venv) akretsis@desktop:~/Desktop$ kubectl -n empyrean get pods
NAME                                          READY   STATUS    RESTARTS   AGE
registry-api-6489f77554-v6q8d                 1/1     Running   0          98s
registry-association-metadata-6455988999-q2lwd   1/1     Running   0          87s
registry-manager-66677cd67-zpxjp              1/1     Running   0          93s
(venv) akretsis@desktop:~/Desktop$
```

Figure 21: EMPYREAN Registry components successful deployment in ICCS's K8s cluster.

The API Gateway exposes a RESTful interface to enable stateless communication and provide external access to Registry's services. This interface is designed to support low-complexity interactions, such as registering new resources, querying available Associations, and retrieving service metadata. Incoming requests to the API Gateway are forwarded to the Registry Manager, which serves as the internal coordination point. The Registry Manager processes requests and interacts with other core components, including the Association Metadata Store, to fetch or update information related to Associations and their registered services.

Figure 22 shows the available REST methods, covering key operations for external entities to interact with the Registry securely and consistently. All endpoints are documented using the OpenAPI specification.

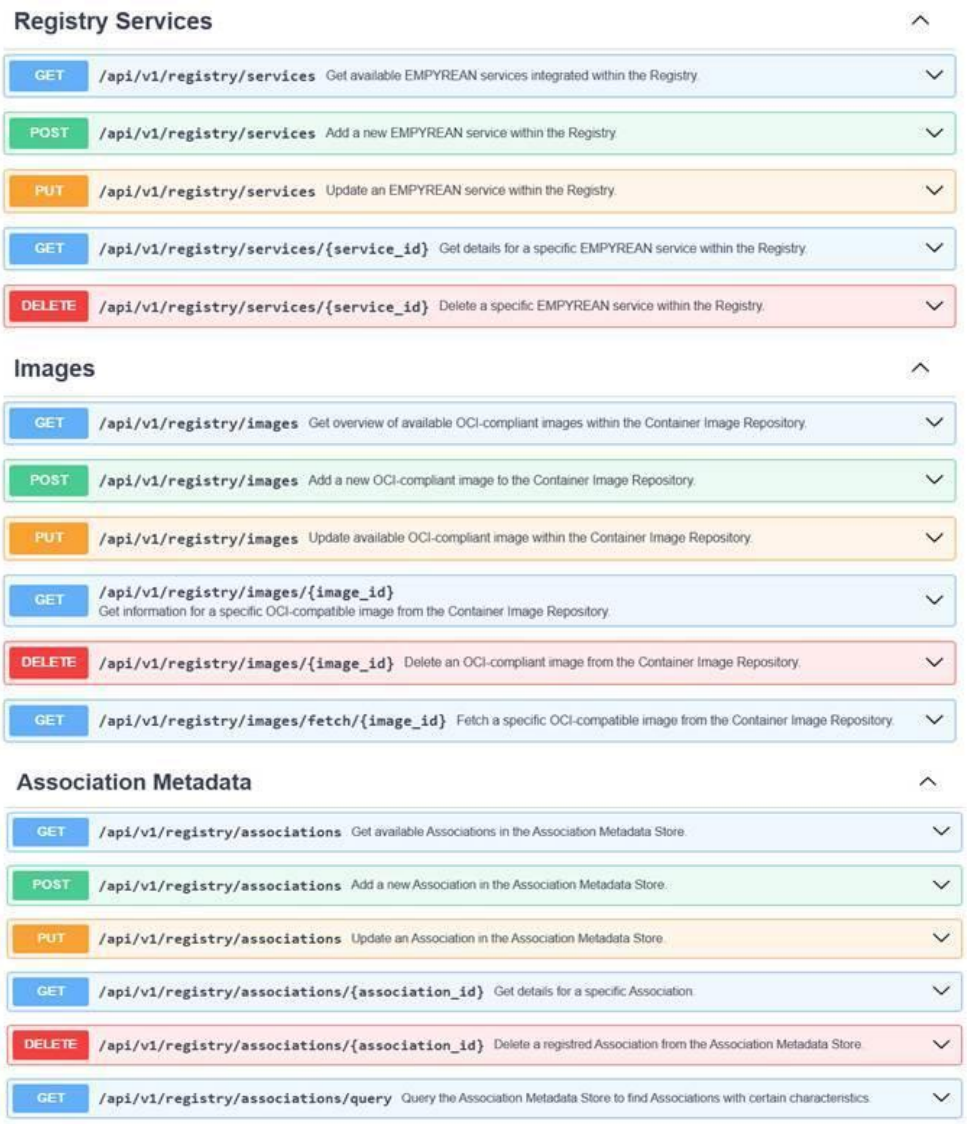


Figure 22: EMPYREAN Registry – API Gateway RESTful API.

The Association Metadata Store aggregates metadata from multiple distributed Associations. Its primary function is to maintain structured information about each Association's participating resources, workload, sharing policies, and other governance rules. This enables effective coordination, trust, and intelligent orchestration across the Association-based continuum. The service provides a RESTful API (Figure 23) to support integration with other EMPYREAN Registry services and EMPYREAN Aggregators.

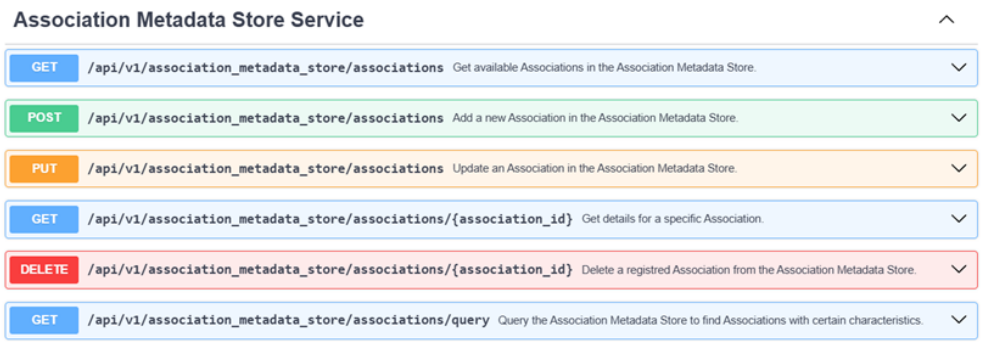


Figure 23: Association Metadata Store RESTful API.

Next, we present key interactions among the components of the EMPYREAN Registry that highlight the stateless REST API exposed by the API Gateway, the role of the Registry Manager, and the internal coordination with services like the Association Metadata Store. In Section 6, we provide a detailed walkthrough of critical operation flows, such as the creation of an EMPYREAN Association, device onboarding, and cross-platform service deployment, which rely on these fundamental Registry operations.

Create a new Association

The definition of a new Association is initiated via a POST request, which is first handled by the API Gateway. The API Gateway (Figure 24) performs initial parameter validation and then forwards the request to the Registry Manager (Figure 25), which orchestrates the creation process by interacting with the Association Metadata Store to persist the new Association record.

```
POST /api/v1/registry/associations

{"name": "ICCS Default Association", "labels": ["platform-arch:amd64", "platform-arch:arm64"],"aggregator_uuid": "a2b66fa3-3f13-416b-8644-56328e0de4ba","owner_uuid": "71c1642b-69c9-4e25-abcc-dda116e8becd","policy_uuid":"5c0eed16-31a8-467d-83db-ef1010466755"}
```

```
INFO:EMPYREAN Registry - API Gateway:Incoming request for new association ...
DEBUG:EMPYREAN Registry - API Gateway:Assigned Association UUID 'f0a55297-d68f-411f-b7fa-9c1526c0db16'
INFO:EMPYREAN Registry - API Gateway:Incoming request to create Association 'ICCS Default Association' with UUID 'f0a55297-d68f-411f-b7fa-9c1526c0db16'
DEBUG:EMPYREAN Registry - API Gateway:{'name': 'ICCS Default Association', 'labels': ['platform-arch:amd64', 'platform-arch:arm64'], 'aggregator_uuid': 'a2b66fa3-3f13-416b-8644-56328e0de4ba', 'owner_uuid': '71c1642b-69c9-4e25-abcc-dda116e8becd', 'policy_uuid': '5c0eed16-31a8-467d-83db-ef1010466755', 'description': '', 'uuid': 'f0a55297-d68f-411f-b7fa-9c1526c0db16', 'status': 'PENDING', 'clusters': [], 'storage_resources': [], 'computing_resources': [], 'schedulable': False, 'logs': [{'timestamp': 1749971871, 'event': 'Create Association object.'}], 'updated_by': 'API.Gateway', 'created_at': 1749971871, 'updated_at': 1749971871}
INFO:EMPYREAN Registry - API Gateway:Notify Registry Manager to handle creation request ...
```

Figure 24: API Gateway – Processing request for creating a new Association.

```
INFO:RegistryManager.RegistryAPIInterface:Service is running ...
DEBUG:RegistryManager.RegistryAPIInterface:Watch event: <etcd3.watch.WatchResponse object at 0x7d60f0f03980>
INFO:RegistryManager.RegistryAPIInterface:Association creation request ...
INFO:RegistryManager.RegistryAPIInterface:Create internal API object
DEBUG:RegistryManager.RegistryAPIInterface:{'name': 'ICCS Default Association', 'labels': ['platform-arch:amd64', 'platform-arch:arm64'], 'aggregator_uuid': 'a2b66fa3-3f13-416b-8644-56328e0de4ba', 'owner_uuid': '71c1642b-69c9-4e25-abcc-dda116e8becd', 'policy_uuid': '5c0eed16-31a8-467d-83db-ef1010466755', 'description': '', 'uuid': 'f0a55297-d68f-411f-b7fa-9c1526c0db16', 'status': 'CREATED', 'clusters': [], 'storage_resources': [], 'computing_resources': [], 'schedulable': False, 'logs': [{'timestamp': 1749971871, 'event': 'Create Association object.'}], 'updated_by': 'API.Gateway', 'created_at': 1749971871, 'updated_at': 1749971871, 'association_uuid': 'f0a55297-d68f-411f-b7fa-9c1526c0db16', 'request': 'POST'}
INFO:RegistryManager.RegistryAPIInterface:Inform Association Metadata Store service for new Association with UUID 'f0a55297-d68f-411f-b7fa-9c1526c0db16'
DEBUG:urllib3.connectionpool:Starting new HTTP connection (1): 147.102.16.114:30086
DEBUG:urllib3.connectionpool:http://127.0.0.1:10086 "POST /api/v1/association_metadata_store/associations HTTP/1.1" 200 2
INFO:RegistryManager.RegistryAPIInterface:Inform corresponding EMPYREAN Aggregator for new Association with UUID 'f0a55297-d68f-411f-b7fa-9c1526c0db16'
DEBUG:urllib3.connectionpool:Starting new HTTP connection (1): 147.102.16.114:30800
DEBUG:urllib3.connectionpool:http://127.0.0.1:10086 "POST /api/v1/aggregator/associations HTTP/1.1" 200 2
INFO:RegistryManager.RegistryAPIInterface:Inform EMPYREAN Storage Service new Association with UUID 'f0a55297-d68f-411f-b7fa-9c1526c0db16'
INFO:RegistryManager.RegistryAPIInterface:Update internal API object for newly created Association with UUID 'f0a55297-d68f-411f-b7fa-9c1526c0db16'
```

Figure 25: Registry Manager – Processing request for creating a new Association.

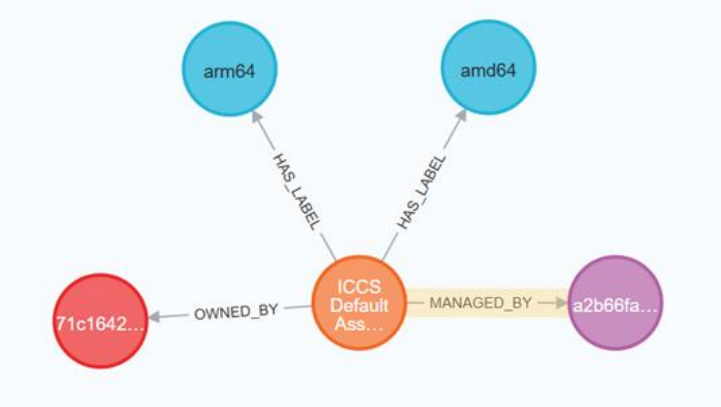


Figure 26: Visual representation of available Associations within Association Metadata Store service of the EMPYREAN Registry

Retrieve all available Associations

When a client issues a request to retrieve information about available Associations, the API Gateway queries the Association Metadata Store through its internal API. The Association Metadata Store handles the retrieval of Association records and returns them to API Gateway.

```
GET /api/v1/registry/associations

[{"name":"ICCS Default Association","uuid":"f0a55297-d68f-411f-b7fa-
9c1526c0db16","aggregator_uuid":"a2b66fa3-3f13-416b-8644-56328e0de4ba","schedulable":
false}]
```

```
DEBUG:urllib3.connectionpool:http://147.102.16.114:30150 "GET /api/v1/registry/associations HTTP/1.1" 200 150
INFO:EMPYREAN Registry - API Gateway:Query Association Metadata Store service to get all available Associations ...
DEBUG:urllib3.connectionpool:Starting new HTTP connection (1): 147.102.16.114:30086
DEBUG:urllib3.connectionpool:http://147.102.16.114:30086 "GET /api/v1/association_metadata_store/associations HTTP/1.1" 200 160
DEBUG:EMPYREAN Registry - API Gateway:[{'schedulable': False, 'name': 'ICCS Default Association', 'uuid': 'f0a55297-d68f-411f-b7fa-9c1526c0db16',
'aggregator_uuid': 'a2b66fa3-3f13-416b-8644-56328e0de4ba'}]
```

Advertise a new service

When a new service is advertised, the API Gateway receives the request and forwards it to the Registry Manager, which validates the service metadata and stores it in the Service Catalogue. This process ensures that the service becomes discoverable across the EMPYREAN platform and is correctly linked to the relevant Association. The following GET method lists the available Aggregator services within the EMPYREAN platform.

```
GET /api/v1/registry/services?category=aggregator

[{"uuid":"a2b66fa3-3f13-416b-8644-
56328e0de4ba","category":"Aggregator","service_endpoint":
"http://147.102.16.114:30800"}, {"uuid":"4790a1e3-aa38-4461-8b81-8f771c60dbb5",
"category": "Aggregator", "service_endpoint": "http://147.102.16.115:30800"}]
```

5.3 Multi-cluster Orchestration Layer

5.3.1 Decision Engine

The Decision Engine Controller coordinates multiple Execution Engines, lightweight, modular workers responsible for executing decision-making logic Figure 27. The Decision Engine interfaces directly with the Service Orchestrator (Section 5.3.2), processing incoming service deployment and management requests. It operates on real-time insights from the EMPYREAN Telemetry Service (Section 5.7.1), allowing it to make data-driven decisions that adapt to changing infrastructure and application conditions. To support distributed, multi-agent orchestration, the Decision Engine leverages the Distributed Data Manager (Section 5.5.2), which is based on Eclipse Zenoh [5]. This integration ensures high-performance, dynamic, and scalable data exchange between orchestration components, even across geographically dispersed clusters.

Further details on the design and implementation of the Decision Engine, including a comprehensive analysis of the initial algorithms developed, are provided in D4.2 (M15) [7].

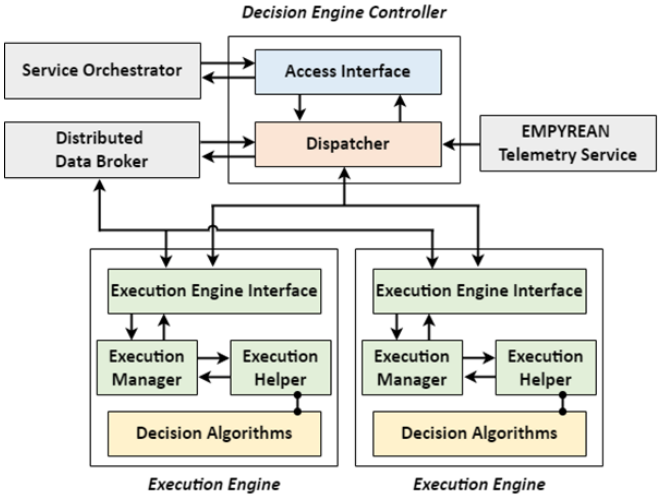


Figure 27: The Decision Engine architecture, main components, and interactions.

Integration and interfaces

The Decision Engine components are implemented in Python and packaged as containerized microservices to ensure streamlined and efficient development, deployment, and testing workflow. Separate container images are provided for the Decision Engine Controller and the Execution Engine, and these are maintained in the official EMPYREAN image repository. Moreover, the corresponding Kubernetes YAML description files are available to facilitate its deployment on Kubernetes platforms. These definitions enable the automatic deployment and scaling of the Decision Engine components. They are also integrated into EMPYREAN's CI/CD pipeline, supporting continuous integration and delivery processes, see Figure 28.

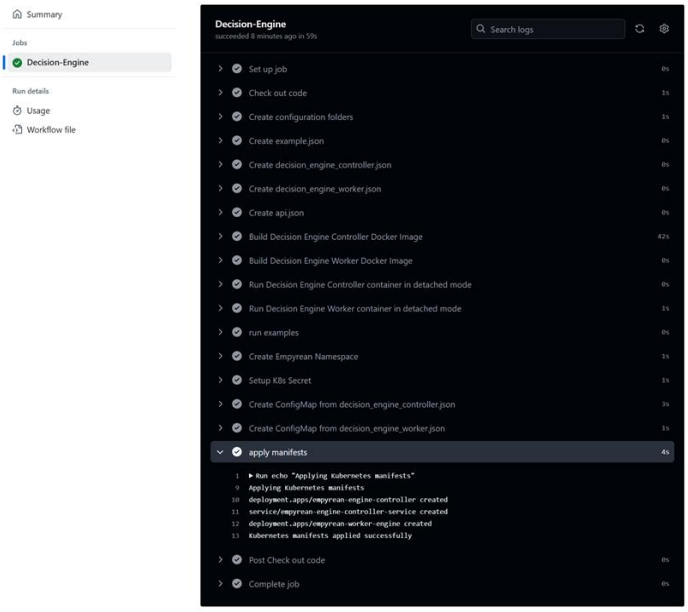
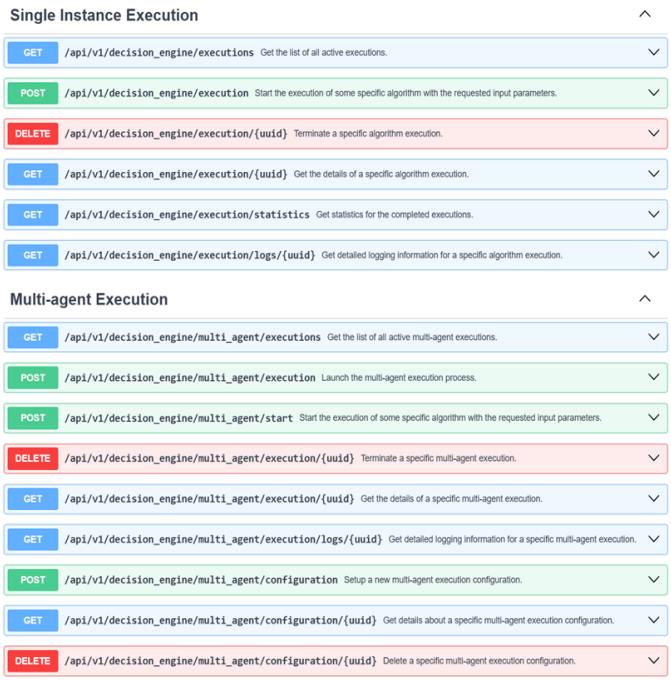


Figure 28: Decision Engine into CI/CD pipeline and deployment in ICCS's K8s cluster.

The Decision Engine exposes two distinct Northbound Interfaces (NBIs) to support interaction with external components and users: a RESTful API and an asynchronous messaging interface based on the Advanced Message Queuing Protocol (AMQP). The REST API interface, see Figure 29, provides a set of control operations that allow external entities to manage and inspect the execution of deployment algorithms, query the status and capabilities of available Execution Engines, and perform administrative tasks related to user management. This interface is fully documented using the OpenAPI specification, ensuring clarity, consistency, and ease of integration for developers and third-party services. Complementing this, the second interface enables asynchronous communication between the Decision Engine Controller and end users, facilitating the exchange of notification messages and results.



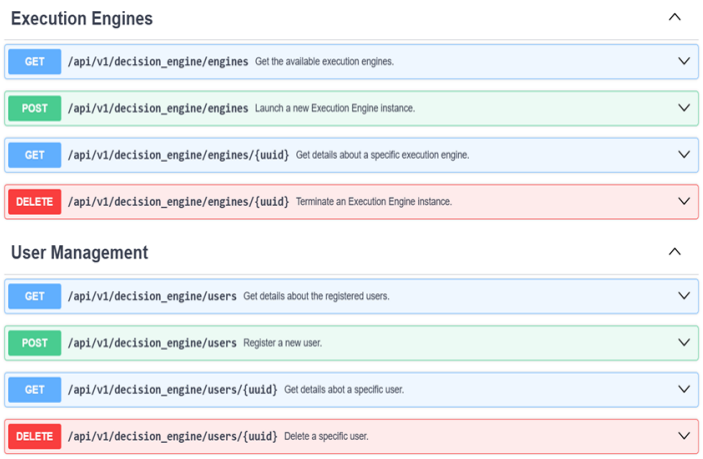


Figure 29: Decision Engine – Access Interface REST API.

Next, we detail the interaction among the Decision Engine components, the Service Orchestrator, and the Telemetry Service for supporting multi-agent operations across EMPYREAN Associations. Specifically, we focus on the initial placement of an application's microservices, considering a scenario involving three available Associations (Figure 30).

This operation corresponds to the operation flow OF 4.1.1, initially defined in D2.3 (M12) and fully specified in D4.2 (M15). It exemplifies how the Decision Engine leverages multi-agent coordination to evaluate resource availability and policy constraints to make an optimal initial placement decision across distributed infrastructure domains abstracted by Associations.

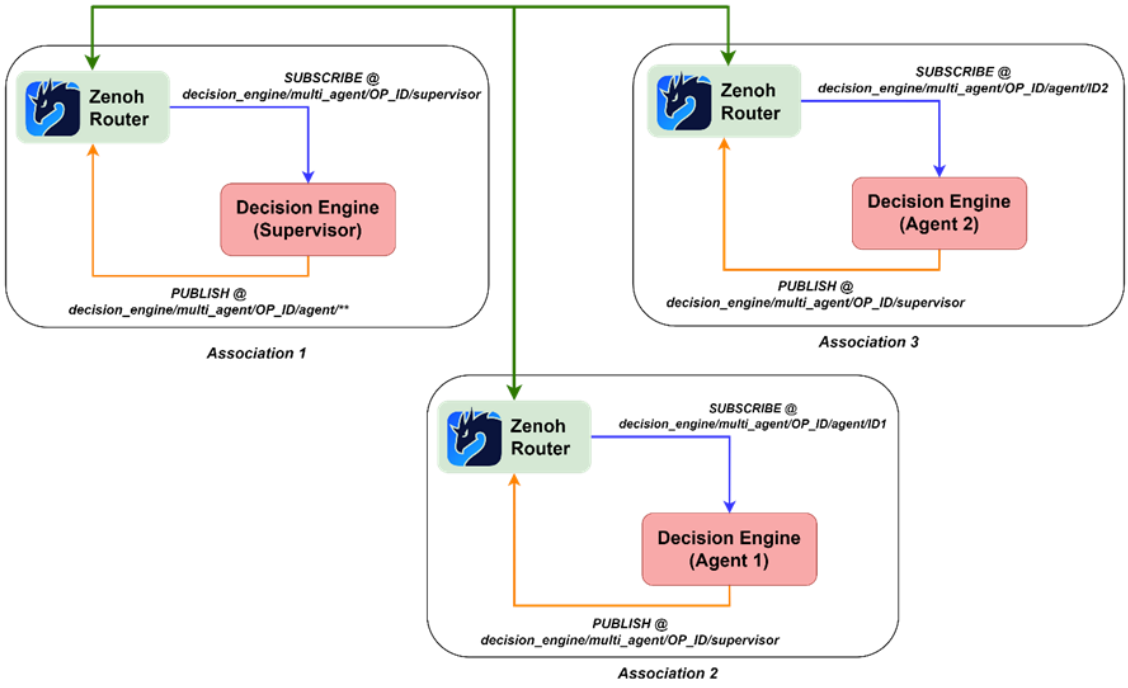


Figure 30: Multi-agent operation across two EMPYREAN Associations for initial placement of application's microservices.

Service Orchestrator to Decision Engine

The Service Orchestrator initiates the multi-agent decision-making process across the Associations by issuing the following POST request to its designated Decision Engine. This request triggers the evaluation of candidate deployment options across multiple infrastructure domains. During this process, the Decision Engine located in the originating Association acts as the supervisor, coordinating the distributed decision-making activity among participating Execution Engines

```
POST /api/v1/decision_engine/multi_agent/execution

{"execution_plugin": "AssociationMatch", "parameters":
{ "application_description":[{"kind":"Deployment","name":"frontend","replicas":1},{"ki
nd":"Deployment", "name": "data-producer","replicas": 1}, {"kind":"Deployment","name":
"classifier", "replicas": 1}, {"kind": "Deployment", "name": "model-manager",
"replicas": 1}, {"kind": "Deployment", "name": "state-manager", "replicas":
1}],"deployment_objectives": [{"colocation": ["frontend", "state-manager"], "data-
producer": {"latency": 5}, "classifier": {"latency": 5, "energy": 3}, "model-manager":
{"availability": 4}}]}}
```

Decision Engine to EMPYREAN Registry

The supervisor Decision Engine queries the EMPYREAN Registry to identify Associations that meet the required deployment criteria, utilizing the Registry's exposed RESTful API.

Figure 31: Query results in the EMPYREAN registry.

Initial coordination among Decision Engines

The Decision Engine dynamically creates exchange topics within the Distributed Data Broker to enable bidirectional communication among participating Decision Engines. It then notifies the selected Decision Engines about the newly created topics by executing the following POST request. The following request corresponds to the configuration of the Decision Engine at the "Association 2".

```
POST /api/v1/decision_engine/multi_agent/configuration
{
  "multi_agent_operation_id ": "984b9759-5a7f-4eb6-8020-0a4e37150893",
  "agent_uuid": "7ee00459-9635-40f9-81fe-ab92be1f3a8c",
  "supervisor_uuid": "299cf3e8-be6c-4ff0-b5c7-7c8eb60be7b6"
}
```

Each selected Decision Engine registers to the assigned topics and sends an acknowledgment message to the supervisor Decision Engine.

```
INFO:DecisionEngine.Engine.EngineInterface:EngineInterface is running ...
INFO:DecisionEngine.Engine.RequestInterface:RequestInterface is running ...
INFO:DecisionEngine.Engine.EngineInstance:Decision Engine "POST /api/v1/decision_engine/multi_agent/configuration HTTP/1.1" 200 -
DEBUG:urllib3.connectionpool:Starting new HTTP connection (1): 47.102.16.114:30020
DEBUG:urllib3.connectionpool:http://147.102.16.114:30020 "POST /api/v1/decision_engine/multi_agent/configuration HTTP/1.1" 200 32
INFO:DecisionEngine.Engine.EngineHelper:Subscribe to Zenoh topic 'decision_engine/multi_agent/984b9759-5a7f-4eb6-8020-0a4e37150893/agen
ts/7ee00459-9635-40f9-81fe-ab92be1f3a8c'
INFO:DecisionEngine.Engine.EngineHelper:Acknowledgment multi-agent supervisor over Zenoh topic 'decision_engine/multi_agent/984b9759-5a
7f-4eb6-8020-0a4e37150893/supervisor/ack'
```

Multi-agent execution

Once all expected acknowledgments are received, the supervisor proceeds to initiate the multi-agent execution by issuing requests through each participating Decision Engine's Access Interface.

```
POST /api/v1/decision_engine/multi_agent/start
```

The supervisor Decision Engine collects responses from the collaborating Decision Engines through the exchange topics provided by the Distributed Data Broker

```
INFO:DecisionEngine.Engine.EngineInstance:Decision Engine "POST /api/v1/decision_engine/multi_agent/start HTTP/1.1" 200 -
DEBUG:urllib3.connectionpool:Starting new HTTP connection (1): 147.102.16.114:30020
DEBUG:urllib3.connectionpool:http://147.102.16.114:30020 "POST /api/v1/decision_engine/multi_agent/start HTTP/1.1" 200 32
INFO:DecisionEngine.Engine.EngineHelper:Start selected algorithm
INFO:DecisionEngine.Engine.EngineHelper:Forward results to multi-agent supervisor over Zenoh topic 'decision_engine/multi_agent/984b975
9-5a7f-4eb6-8020-0a4e37150893/supervisor/results'
```

Using the received data, it determines the optimal distribution of the application's microservices across the available Associations, aiming to meet user requirements while maximizing resource efficiency.

Multi-agent termination

At the end of the process, the supervisor Decision Engine notifies the participating Decision Engines of the multi-agent session completion by sending a termination request through their exposed RESTful API. For example, to notify the Decision Engine in Association 3, the following DELETE request is executed.

```
INFO:DecisionEngine.Engine.EngineInstance:Decision Engine "DELETE /api/v1/decision_engine/multi_agent/configuration/984b9759-5a7f-4eb6-
8020-0a4e37150893 HTTP/1.1" 200 -
DEBUG:urllib3.connectionpool:Starting new HTTP connection (1): 147.102.16.114:30020
DEBUG:urllib3.connectionpool:http://147.102.16.114:30020 "DELETE /api/v1/decision_engine/multi_agent/configuration/984b9759-5a7f-4eb6-8
020-0a4e37150893 HTTP/1.1" 200 32
INFO:DecisionEngine.Engine.EngineHelper:Receive termination request
INFO:DecisionEngine.Engine.EngineHelper:Helper Instance ID '2' finished.
```

5.3.2 Service Orchestrator

The Service Orchestrator operates within an EMPYREAN Association, coordinating multiple platform-specific container orchestration systems such as Kubernetes (K8s) and Lightweight Kubernetes (K3s). These platform-level orchestrators manage distinct infrastructure segments within the Association. To enable abstracted and unified service deployment and management, the Service Orchestrator employs hierarchical and distributed orchestration mechanisms, which operate at the Association level. This approach ensures consistent and policy-compliant service management across diverse platforms.

The Service Orchestrator consists of two primary services: the Orchestration API Server and the Orchestration Manager. Together, these services form the core of the orchestration logic and expose the interfaces necessary for service deployment and management within an EMPYREAN Association. The architecture is complemented by a centralized Datastore that ensures consistent operation and coordination among the various Service Orchestrator components.

Supporting this architecture, EMPYREAN Controllers, also referred to as Orchestration Drivers, are deployed on each managed platform. These components are responsible for interfacing with the Local Orchestrators and underlying platform-specific APIs, handling low-level orchestration operations such as container scheduling, resource provisioning, and telemetry collection. Figure 32 depicts the architecture and the main components of the Service Orchestrator and EMPYREAN Controller.

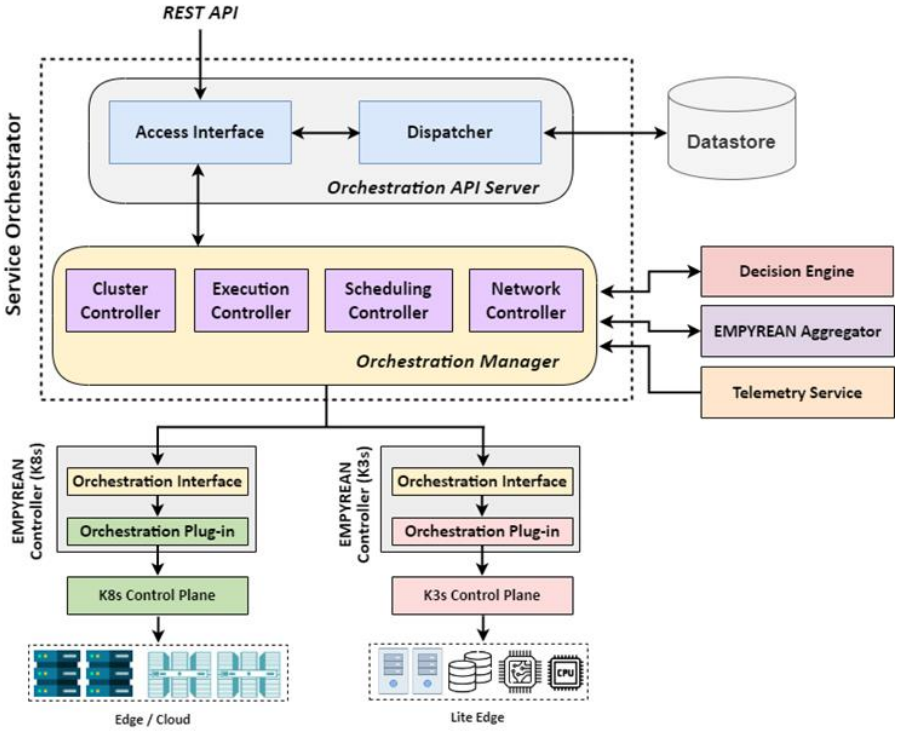


Figure 32: Service Orchestrator and EMPYREAN Controller architecture and its main components.

Integration and interfaces

The Resource Orchestrator services, and associated Orchestration Drivers have been developed in Python. These components, along with their configuration files, are packaged as modular Python applications and seamlessly integrated into the EMPYREAN CI/CD pipeline. The Orchestration API and Orchestration Manager services are bundled into a dedicated container image, while both Orchestration Drivers are distributed via a shared image.

To support Kubernetes-based deployment, all necessary Kubernetes YAML manifests, including ConfigMap, Deployment, Service, and Ingress definitions, have been created. These descriptions facilitate consistent, reproducible deployment of the orchestration components across test and production environments. Furthermore, the entire orchestration stack has been integrated into EMPYREAN's CI/CD pipeline, enabling automated deployment into the EMPYREAN testbed infrastructure (Figure 33).

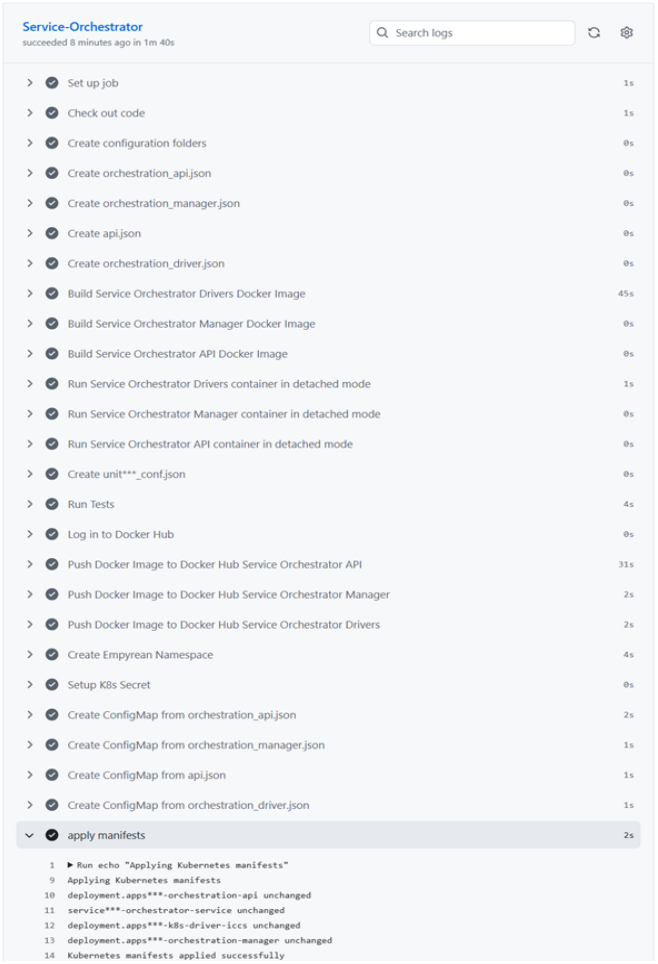


Figure 33: Service Orchestrator and EMPYREAN Controller into EMPYREAN CI/CD pipeline and their successful deployment in ICCS's K8s cluster.

A comprehensive overview of the setup used for initial integration testing of the Service Orchestrator is presented in Figure 34, showcasing the robustness, modularity, and integration readiness of the developed solution.

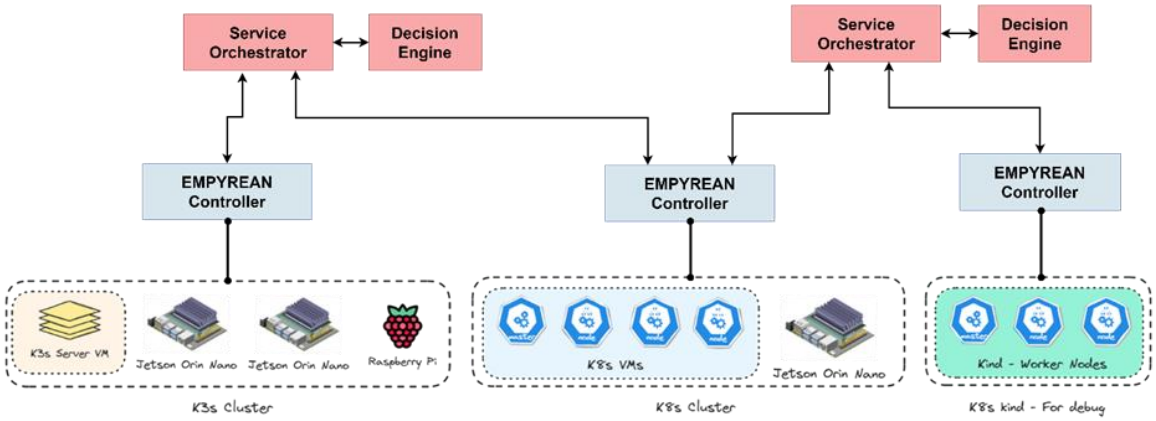


Figure 34: Setup for integration tests of Service Orchestrator and EMPYREAN Controllers.

The initial version of the exposed REST API includes several methods organized into two main categories. The first set of methods, see Figure 35, enables the deployment and management of cloud-native applications within and across Associations and the cognitive creation of secure storage policies.

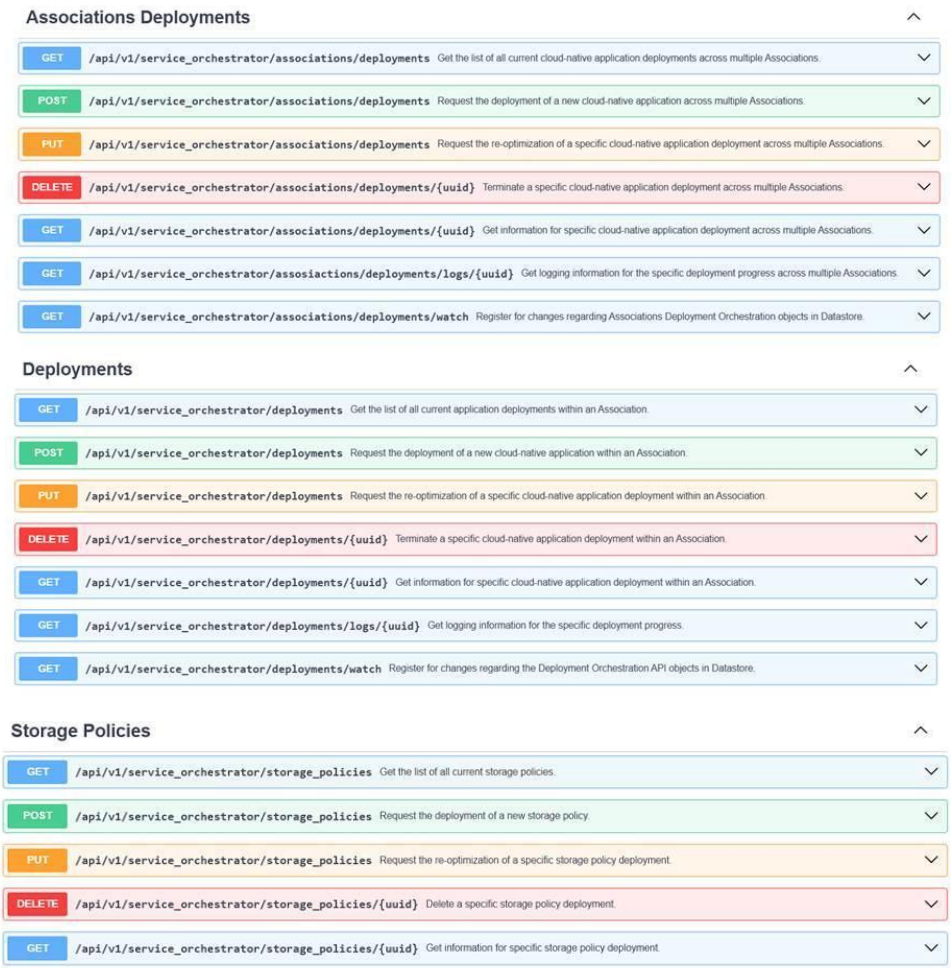


Figure 35: Service Orchestrator REST API.

The second set, see Figure 36, abstracts the interaction of the Orchestration Manager and EMPYREAN Controllers services.

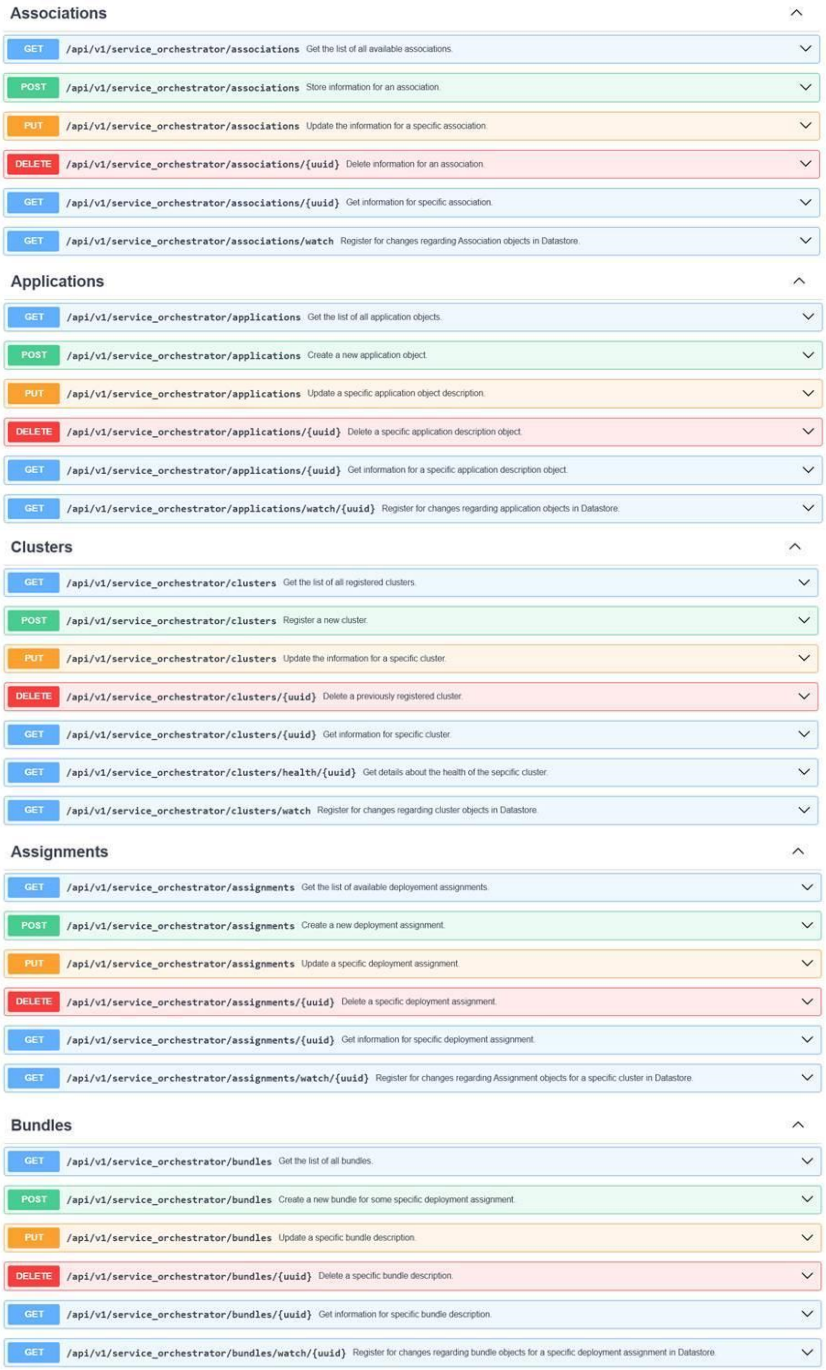


Figure 36: Service Orchestrator REST API – Methods related to inter-component communication.

Service Orchestrator and EMPYREAN Controllers

During initialization, each EMPYREAN Controller registers with its associated Service Orchestrator by using a REST endpoint provided by the Orchestration API Server. As part of this process, it also sends a summary of the resources available on the cluster it manages.

```
PUT /api/v1/service_orchestrator/clusters
```


```
{"cluster_uuid": "2b05cfdf-679f-45f1-95f8-a334ec87faaf", "type": "k8s", "info":
{"nodes": [{"node_name": "master-amorgos", "labels": {"beta.kubernetes.io/arch":
"amd64", "kubernetes.io/hostname": "master-amorgos"}, "capacity": {"cpu": "2",
"ephemeral-storage": "60575596Ki", "hugepages-2Mi": "0", "memory": "8132592Ki",
"pods": "110"}, {"node_name": "wn1-ios", "labels": {"beta.kubernetes.io/arch":
"arm64", "jetson": "true", "kubernetes.io/hostname": "wn1-ios"}, "capacity": {"cpu":
"6", "ephemeral-storage": "30538800Ki", "hugepages-1Gi": "0", "hugepages-2Mi": "0",
"hugepages-32Mi": "0", "hugepages-64Ki": "0", "memory": "7800596Ki", "pods":
"110"}}]}}
```


The Orchestration API Server uses this data to update the corresponding entries in the Datastore. Below is an example showing the K8s and K3s clusters managed by the first Service Orchestrator in the testbed setup illustrated above. This information can be retrieved using the following GET request.

```
GET /api/v1/service_orchestrator/clusters
```


```
{"clusters":[{"cluster_uuid":"2b05cfdf-679f-45f1-95f8-
a334ec87faaf","type":"k3s","last_seen":"1750059454"}, {"cluster_uuid":"7628b895-3a91-
4f0c-b0b7-033eab309891","type":"k8s","last_seen":"1750059501"}]}
```


Cross-Association application deployment

Next, we examine the integration among the Service Orchestrator services during the deployment of a cloud-native application. Specifically, we consider a demo application composed of five microservices: “frontend”, “state-manager”, “model-manager”, “classifier”, and “data-producer”. The deployment process begins when the Orchestration API Server receives a request through its POST endpoint (/api/v1/service_orchestrator/deployments). Upon receiving the request, the API Server validates the payload and creates the corresponding Deployment object in the Datastore, initializing the orchestration workflow


```
{"kind": "Deployment", "name": "demo-deployment ", "deployment_uuid": "585fdf45-4959-
f4e1-4d43-be89830bd890", "deployment_description": "YAML DESCRIPTION", "assignments":
[], "assignments_status": [], logs":[{"timestamp":1750060830, "event":"Deployment
description received."}], "status":1, "updated_by": "Orchestration.API","created_at":
1750060830, "updated_at": 1750060830 }
```


The Orchestration Manager is subsequently notified of the new deployment request and, through its Scheduler Controller, invokes the Decision Engine to produce a high-level orchestration decision, defining how the application's microservices should be distributed across the available platforms (Section 5.3.1). Next, we present the output generated by the Decision Engine, which guides the subsequent execution phase of the deployment process.


```
{"kind":"Deployment", "assignments":[{"cluster_uuid": "2b05cfdf-679f-45f1-95f8-
a334ec87faaf", "deployments":["data-manager","classifier","model-manager"]},
{"cluster_uuid":"7628b895-3a91-4f0c-b0b7-033eab309891", "deployments":["state-
manager","frontend"]}]}
```

Based on the orchestration decision provided by the Decision Engine, the Execution Controller of the Orchestration Manager proceeds to create the corresponding Assignment and Bundle objects in the Datastore. Two Assignment objects (UUIDs "*d99b982d-d558-4c06-bdcf-6279d037d5ff*" and "*03b61ed2-91b4-478c-8842-9ced0f8bc11b*") are created to reflect the allocation of the application's microservices across the two selected platforms. These Assignment objects are linked to the initial Deployment object with UUID *"585fdf45-4959-f4e1-4d43-be89830bd890*", enabling full traceability of the deployment workflow. Log messages from the Orchestration Manager and Datastore confirming these operations are included below. In parallel, five Bundle objects are generated, each representing a single microservice of the application. These Bundles are associated with the relevant Assignment object and contain all necessary deployment descriptors for execution at the platform level.

```
INFO:Orchestrator.OrchestrationManager.OrchestrationAPIInterface:OrchestrationAPIInterface service is ready ...
INFO:Orchestrator.OrchestrationManager:OrchestrationManager service is ready ...
INFO:Orchestrator.OrchestrationManager:Orchestration Manager is ready ...
INFO:Orchestrator.OrchestrationManager.OrchestrationAPIInterface:Deployment event(s) ...
INFO:Orchestrator.OrchestrationManager.OrchestrationAPIInterface:Deployment event for key '/service/orchestrator/deployments/deployment/585fdf45-4959-f4e1-4d43-be89830bd8
90'
INFO:Orchestrator.OrchestrationManager:Handle deployment request ...
DEBUG:urllib3.connectionpool:Starting new HTTP connection (1): 147.102.22.140:10100
DEBUG:urllib3.connectionpool:http://147.102.16.114:30100 "GET /api/v1/orchestrator/clusters?active=15m&details=1 HTTP/1.1" 200 4036
DEBUG:urllib3.connectionpool:Starting new HTTP connection (1): 147.102.16.114:30100
INFO:Orchestrator.OrchestrationManager.OrchestrationAPIInterface:Deployment event(s) ...
DEBUG:urllib3.connectionpool:http://147.102.16.114:30020 "POST /api/v1/decision_engine/execution HTTP/1.1" 200 76
INFO:Orchestrator.OrchestrationManager.DecisionEngineDriver:Receive Decision Engine response for execution uuid '6cc91bcf-e7e7-4330-ac1f-e711da1edb89'
DEBUG:Orchestrator.OrchestrationManager.DecisionEngineDriver:{"kind": "Deployment", "assignments": [{"cluster_uuid": "2b05cfdf-679f-45f1-95f8-a334ec87faaf", "deployments"
: ["data-manager", "classifier", "model-manager"]}, {"cluster_uuid": "7628b895-3a91-4f0c-b0b7-033eab309891", "deployments": ["state-manager","frontend"]}]}
DEBUG:Orchestrator.OrchestrationManager.OrchestrationAPIInterface:Create Bundle '9d1fc097-dec9-f438-83ea-625278519ca9' in ETCD
DEBUG:Orchestrator.OrchestrationManager.OrchestrationAPIInterface:Create Bundle '351e09cd-a42b-469b-81bc-935bb43b5282' in ETCD
DEBUG:Orchestrator.OrchestrationManager.OrchestrationAPIInterface:Create Bundle 'fe1ac76c-305c-4096-bd15-af6459e1884d' in ETCD
DEBUG:Orchestrator.OrchestrationManager.OrchestrationAPIInterface:Create Bundle 'a2d24afd-2691-4a5d-97be-10c9b9533786' in ETCD
DEBUG:Orchestrator.OrchestrationManager.OrchestrationAPIInterface:Create Bundle '8a3204cf-8ebf-4aa0-b667-b7078d7f5796' in ETCD
DEBUG:Orchestrator.OrchestrationManager.OrchestrationAPIInterface:Create Assignment 'd99b982d-d558-4c06-bdcf-6279d037d5ff' for cluster '7628b895-3a91-4f0c-b0b7-033eab3098
91' in ETCD
DEBUG:Orchestrator.OrchestrationManager.OrchestrationAPIInterface:Associate Assignment 'd99b982d-d558-4c06-bdcf-6279d037d5ff' with Bundles ['9d1fc097-dec9-f438-83ea-62527
8519ca9', '351e09cd-a42b-469b-81bc-935bb43b5282']
DEBUG:Orchestrator.OrchestrationManager.OrchestrationAPIInterface:Create Assignment '03b61ed2-91b4-478c-8842-9ced0f8bc11b' for cluster '2b05cfdf-679f-45f1-95f8-a334ec87fa
af' in ETCD
DEBUG:Orchestrator.OrchestrationManager.OrchestrationAPIInterface:Associate Assignment '03b61ed2-91b4-478c-8842-9ced0f8bc11b' with Bundles ['fe1ac76c-305c-4096-bd15-af645
9e1884d', 'a2d24afd-2691-4a5d-97be-10c9b9533786','8a3204cf-8ebf-4aa0-b667-b7078d7f5796']
DEBUG:Orchestrator.OrchestrationManager.OrchestrationAPIInterface:Update Deployment '585fdf45-4959-f4e1-4d43-be89830bd890' in ETCD
INFO:Orchestrator.OrchestrationManager.OrchestrationAPIInterface:Deployment event(s) ...
INFO:Orchestrator.OrchestrationManager.OrchestrationAPIInterface:Deployment event(s) ...
```

Figure 37: Orchestration Manager log messages while processing Decision Engine response for the application placement.

Orchestration Manager and EMPYREAN Controllers

The creation of Assignment objects triggers the activation of the EMPYREAN Controllers on the two target platforms. Each Controller receives its respective Assignment object, which contains references to the associated Bundles. Upon receipt, the EMPYREAN Controller fetches the content of each referenced Bundle from the Datastore and leverages the underlying K8s/K3s API to perform the required deployment actions on its local infrastructure.

Next, we present log messages from the EMPYREAN Controller operating on the ICCS Kubernetes cluster (7628b895-3a91-4f0c-b0b7-033eab309891), confirming the successful execution of the assigned Bundle (Figure 38).

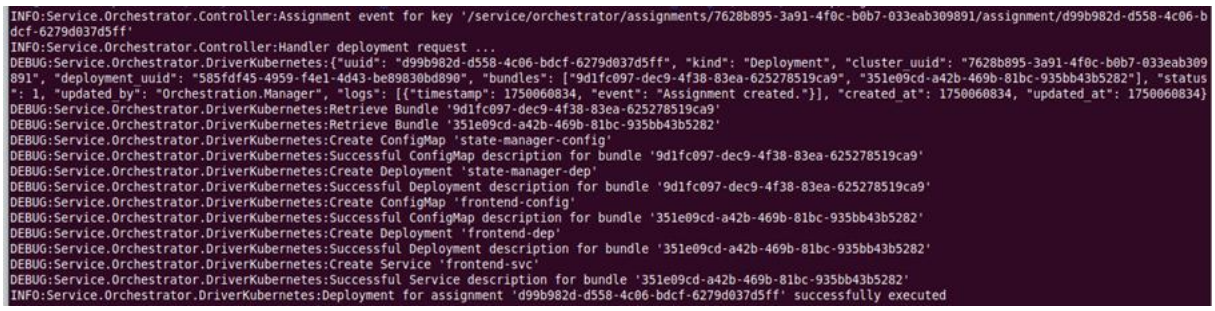


Figure 38: Log messages from the EMPYREAN Controller operating on the ICCS Kubernetes cluster.

5.4 Resource Management Layer

5.4.1 AI-Enabled Workloads Autoscaling

In D5.2, we rebuilt the metrics dataflow in Ryax scheduler and enhanced its interaction with standardised metrics endpoints used by the AI-enabled autoscaler, the Intelliscale.

Previously, the metrics gathering logic was tightly coupled with the autoscaler pod. This functionality has now been decoupled and reimplemented as a general-purpose component within the Ryax Worker. This redesign enables broader reuse across other modules, such as the accounting service, UI, and data persistence layer, facilitating the generation of execution datasets for training purposes. The metrics collection and computation logic was restructured to prioritize both speed and accuracy. Additionally, we standardized all metrics sources to use a homogeneous Prometheus-compatible API, simplifying future development and extensions.

The interaction between the metrics gathering component and Intelliscale has also been significantly improved to be lighter, more flexible, and fine-grained. Rather than transmitting entire histograms, only a single summary datapoint per execution is now sent to the Intelliscale model. Communication has shifted from passive polling to proactive updates, eliminating the previous 5-minute cold start delay. As a result, the Intelliscale pod is now dedicated solely to executing AI algorithms, allowing it to independently support larger AI-enable autoscaling models without interfering with the time-sensitive Ryax scheduler.

Metrics gathering component: Integration and Interfaces

The metrics gathering system is a core part of the Ryax Worker, Ryax's distributed component deployed at each site. It consists of four main subcomponents: a metrics endpoint fetcher, metrics calculator, lifecycle manager, and downstream data feeder, each contributing to efficient and reliable metrics collection and delivery.

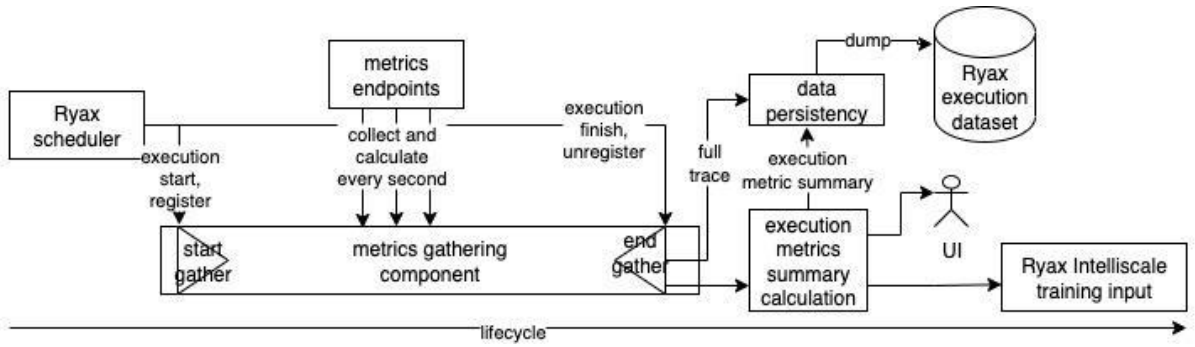


Figure 39: Lifecycle of metrics gathering for an incoming execution.

Figure 39 shows the complete lifecycle of metrics gathering for an incoming execution. A long-running loop fetches and parses metrics from all available metrics endpoints, activated only when at least one execution is registered. When an execution starts, it is registered into this loop, enabling real-time extraction and in in-memory storage of relevant metrics. When the execution ends, it is unregistered from the loop, all the related traces are fed to downstream components, and the corresponding memory is freed.

Downstream components include a metrics summary calculator that computes key per-execution level metrics, including percentiles, averages, and maximum utilization values. Both the raw traces and the summaries are feed to (i) UI to let user know the utilization, (ii) the accounting component for tracking and resource auditing, (iii) data persistency for long-term storage and future analysis, and (iv) Intelliscale for feeding AIU models used in autoscaling.

Metrics are collected from Prometheus-compatible sources to ensure format consistency and extensibility. These include: (i) cAdvisor Prometheus endpoint for CPU, RAM, network, and disk metrics, and (ii) NVIDIA DCGM Exporter for GPU and Multiple-Instance GPU (MIG) metrics. This approach replaces previously heterogeneous sources like the Kubernetes client API (which provide responses in JSON format), simplifying the calculation logic and improving its expandability to new metric sources. Due to the large size of these Prometheus metrics, a pre-filtering mechanism is applied before parsing them. This skips irrelevant lines, reducing computation overhead and improving performance.

Below are the samples of cAdvisor and DCGM metrics API in Prometheus format:

```
container_cpu_usage_seconds_total{container="",cpu="total",id="/kubepods.slice/kubepod
s-besteffort.slice/kubepods-besteffort-
pod1dd6c946_7554_46f7_84b4_852924e0429c.slice",image="",name="",namespace="ryaxns-
monitoring",pod="prometheus-prometheus-node-exporter-wxjfw"} 4767.442561 1742249899112

DCGM_FI_DEV_FB_USED{gpu="0",UUID="GPU-e7fe0866-4c61-ca8e-9094-
d8e15a074343",device="nvidia0",modelName="NVIDIA H100
PCIe",GPU_I_PROFILE="1g.10gb",GPU_I_ID="7",Hostname="scw-k8s-mlintra-pool-gpu-
138004ab087a43d2a62a2",DCGM_FI_DRIVER_VERSION="550.54.14",container="",namespace="",po
d=""} 12
```

After fetching metrics in Prometheus format, a dedicated parsing component processes the data. Many key metrics are not directly available from the raw endpoints. For example, the CPU utilization is the total rate of the container's CPU usage in seconds. Typically, in

Prometheus, we use the “irate” function to obtain the CPU utilization, but here we implement a simplified yet accurate “irate” in the metrics parser. This is more adapted and ignores some corner cases that will never be used in our gathering logic. This approach also avoids the delay and extra consumption associated with introducing the entire Prometheus stack. We only need a tiny part of the Prometheus logic, so introducing the whole software just for this is not good.

$$\text{rate} = \frac{\text{starting value in timerange (v2)} - \text{next occurance value in timerange (v1)}}{\text{starting time in timerange (t2)} - \text{next occurance time in timerange (t1)}}$$

API interaction with Ryax Workflow Management and AI-Enabled Workload Autoscaling (Intelliscale)

After obtaining the full execution trace and summary metrics are collected, the Intelliscale AI autoscaling model receives input from both the metrics summary and additional information about the execution obtained from Ryax scheduler.

The interaction is implemented via a gRPC-based API, which defines the expected metrics fields needed by Intelliscale. Among all these input fields, the Ryax scheduler provides contextual information about each execution, enhancing the autoscaling model’s decision-making capabilities. Figure 40 shows part of the metrics summary provided by metrics gathering component, including additional execution data supplied by the scheduler.

```
string execution_id = 1;
string execution_user_inputs = 2;
string workflow_id = 3;
uint32 action_order = 4;
string action_container_image = 5;
string action_run_id = 6;

// Metrics
ExecutionState execution_final_state = 7;
float execution_time_seconds = 8;
```

```
optional float avg_gpu_mem_util_mb = 24;
optional float avg_cpu_util_cores = 25;
optional float avg_memory_util_bytes = 26;
optional float avg_network_receive_bytes_per_sec = 27;
optional float avg_network_transmit_bytes_per_sec = 28;

optional float avg_disk_fs_read_bytes_per_sec = 29;
optional float avg_disk_fs_write_bytes_per_sec = 30;
optional float avg_parallel_executions_on_same_gpu = 31;
optional float sp_90_cpu_util_cores = 32;
optional float sp_90_memory_util_bytes = 33;

optional float sp_95_cpu_util_cores = 34;
optional float sp_95_memory_util_bytes = 35;
optional float sp_98_cpu_util_cores = 36;
optional float sp_98_memory_util_bytes = 37;
```

Figure 40: Metrics summary from metrics gathering process.

5.4.2 Unikernel Deployment

In modern cloud environments, generic containers are widely used for their flexibility and ease of orchestration, providing isolated user-space environments atop shared host kernels. Sandboxed containers take this further by enforcing stronger security boundaries, often leveraging lightweight hypervisors or additional isolation mechanisms to reduce the host's attack surface. However, unikernels, specialized, single-address-space machine images tailored for a specific application, offer even tighter integration and performance but have historically been challenging to manage using standard cloud-native tools. To bridge this gap, the urunc runtime is introduced as a solution that harmonizes the deployment and management of unikernel workloads within cloud-native ecosystems. By providing a unified interface, urunc allows both traditional containerized applications and unikernel instances to coexist and interoperate seamlessly, enabling organizations to harness the security and efficiency benefits of unikernels without sacrificing the agility and tooling ecosystem of modern container platforms.

Container Runtime

The urunc runtime emerges as an innovative bridge between traditional containerized workloads and unikernel-based applications within cloud-native environments. Unlike standard approaches that force organizations to choose between familiar container interfaces and the performance or security benefits of unikernels, urunc enables both paradigms to coexist on a unified platform. By doing so, it allows operators to leverage unikernels, lightweight, highly specialized single-address-space machine images, for use cases where minimal attack surface and maximum efficiency are crucial, all without forfeiting the operational simplicity or orchestration capabilities of conventional container mechanisms.

Central to urunc's integration into the cloud-native ecosystem is its implementation of the Container Runtime Interface (CRI), the standardized protocol used by Kubernetes and similar orchestrators to communicate with container runtimes. The CRI compatibility ensures that urunc can accept workload scheduling, lifecycle management, and resource allocation requests just like any mainstream container runtime. This facilitates seamless deployment and management across heterogeneous workloads, empowering users to run, update, and monitor unikernel instances alongside standard Linux containers through the same set of tools and workflows. As a result, urunc not only expands the deployment possibilities for next-generation cloud applications but also preserves the developer and operator experience, accelerating the adoption of unikernels in practical, production-ready environments.

Coupled with our unikernel builder (bunny), described in Section 5.1.4, in EMPYREAN we are able to build and deploy single-application kernels (or unikernels) in a pure cloud native way. In the code snippets below, we provide the process to build a simple nginx (webserver) using bunny (Table 6, Table 7), and the process to deploy it in K8s, using our container runtime, urunc (Table 8).

Table 6: Nginx Bunnyfile

```
#syntax=harbor.nbfc.io/nubificus/bunny:0.0.2
version: v0.1

platforms:
  framework: linux
  monitor: qemu
  # monitor: firecracker
  architecture: x86

rootfs:
  from: docker.io/library/nginx:alpine
  type: raw

kernel:
  from: local
  path: vmlinuz

cmdline: "/usr/sbin/nginx -g \"daemon off; error_log stderr debug;\""
```

Table 7: Build urunc container image and push to a generic container image registry

```
docker build -f bunnyfile -t harbor.nbfc.io/nubificus/urunc/nginx-linux-urunc:x86_64 .
docker push harbor.nbfc.io/nubificus/urunc/nginx-linux-urunc:x86_64
```

Table 8: K8s manifest to deploy a urunc-compatible image

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    run: nginx-urunc
  name: nginx-urunc
spec:
  replicas: 1
  selector:
    matchLabels:
      run: nginx-urunc
  template:
    metadata:
      labels:
        run: nginx-urunc
    spec:
      runtimeClassName: urunc
      containers:
      - image: harbor.nbfc.io/nubificus/urunc/nginx-linux-urunc:x86_64
        imagePullPolicy: Always
        name: nginx-urunc
        ports:
        - containerPort: 80
          protocol: TCP
        resources:
          requests:
            cpu: 10m
      restartPolicy: Always
---
apiVersion: v1
kind: Service
metadata:
```

```
  name: nginx-urunc
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    run: nginx-urunc
  sessionAffinity: None
  type: ClusterIP
```

5.4.3 Hardware Acceleration Abstractions

The vAccel framework enables seamless acceleration workload offloading by abstracting hardware acceleration capabilities and offering a unified API for diverse backend targets. In the context of EMPYREAN, vAccel is integrated to support efficient, low-latency execution of AI/ML workloads across distributed, heterogeneous environments.

A key aspect of this integration involves supporting libRRR, the RDMA-capable, user-level communication library described in D3.2 and Section 5.5.1, as a vAccel transport plugin. This integration enables seamless remote AI/ML inference task offloading to hardware-accelerated endpoints across the EMPYREAN IoT-edge-cloud continuum. By embedding the vAccel execution model within libRRR, EMPYREAN introduces a lightweight, low-latency, and high-performance mechanism for invoking vAccel plugins over RDMA channels, ensuring optimal performance and scalability.

An initial integration is in place, providing increased Frames Per Second, for a YOLOv8 object detection example, running on pyTorch (stock, vAccel local, vAccel over RDMA).

5.5 Data Management and Interconnection Layer

5.5.1 Software Defined Edge Interconnect

The EMPYREAN software-defined interconnect provides RDMA transport services to EMPYREAN components that require low-latency and/or high-bandwidth communication, such as the disaggregated vAccel framework developed by NUBIS.

This software-defined interconnect has two main subsystems: (i) the RDMA datapath and the (ii) the software-defined control plane that controls and authenticates the RDMA pairings between actors.

In this initial release, we developed a full-fledged RDMA datapath solution and completed the first integration with vAccel. Implementation of the software-defined control plane is scheduled for the next implementation iteration of the period.

EMPYREAN's software-defined RDMA service is wrapped in a C library that is named RDMA Remote Ring or "triple-R" (librrr). The basic concept, depicted in Figure 41, centers around a ring buffer-based structure along with the provision of a very friendly software I/O dispatch interface.

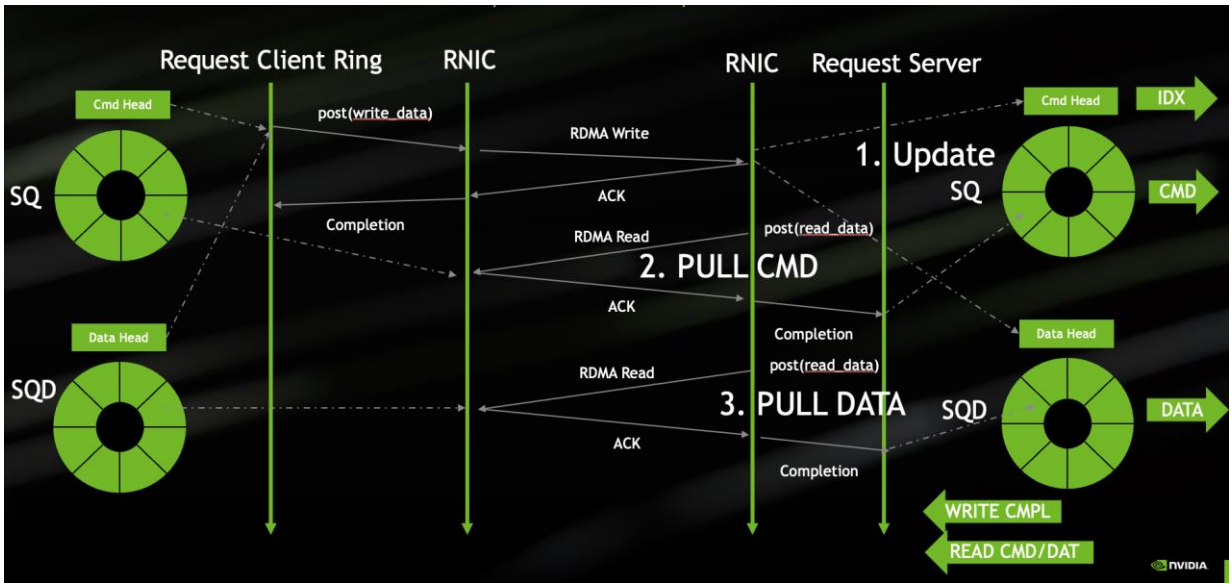


Figure 41: Overview of software-defined RDMA service operation.

In this model, the producers of the ring add requests and data to local in-memory ring buffer structures, while the remote consumers get these requests and data from their corresponding local in-memory rings. LibRRR transparently synchronizes the described disaggregated ring state by detecting aggregation opportunities and efficiently leveraging RDMA.

The figure shows how RDMA NICs (RNICs) are instructed by librrr to update the ring data and head/tail pointers in a coordinated manner. The ring interface is highly effective for asynchronous I/O, where the request path is completed decoupled from the response path and can be implemented using independent threads. The rings also offer backpressure mechanisms, and with properly selected ring buffer sizes, the system adapts I/O performance to match either full link capacity or expose endpoint data generation or ingestion bottlenecks that do not allow the full link bandwidth to be utilized.

Librrr has a C software interface which is comprised of functions for: (i) initialization and setup of the RDMA resources, (ii) registration of ring buffer constructs and in-memory resources with the network, (iii) ring buffer APIs for commands and data tuples to handle to manipulate ring data (produce, consume and synchronize ring data as required, and (iv) tear down and graceful disconnection that that cleans up also the resources.

This simple interface offers transparently RDMA services to user with minimal overhead but also comes with responsibilities that relate to standard circular buffer management. Incorrect handling of circular buffers within program flow may lead to deadlocks. Although librrr does not offer any build-in support to identify deadlocks that arise from using the API with

improper sequence, the library comes with guidelines and recommendations for good practices. Figure 42 presents the set of C functions currently provided by the librrr.

```c
/*
 * rrr_init_rdma_connection: initiates rdma connection over TCP tunnel. It does not depend on rdma_cm it implements it's own simplified version.
 * if server_name is NULL this enters a server mode else it is client mode. This function fills up all rdma_connection struct fields with properly
 * initiated rdma attributes. It keeps the tcp_tunnel active and returns a sockfd. It needs to be reinvoked if more rdma connections need to be set.
 * The server is always the responder and the initiator is always the client.
 * */
int rrr_init_rdma_connection(struct rdma_connection *res, int sockfd, const  char * dev_name, unsigned ib_port);

/*
 * rrr_init_rings: it initiates  SQ/CQ mirror pair for both cmd and data. It needs as arguments the local receive and send buffer address arrays,
 * which are expected to be already initialized externally. For the cmd/cmpl buffers it needs the size of the structs that will be mapped. These structs   need to be packed.
 */
int rrr_init_rings(int sockfd, struct rdma_connection *res, struct rrr_ring_pair *ringp, char * recv_bufs[], char * send_bufs[],\
                                int blocksize, int iodepth, int submit_ctrl_buf_size, int completion_ctrl_buf_size, bool isServer, bool ext_buf_management);

/**
 * Get the next head entry of the command ring.This is for adding data. Upon success it returns a pointer to the cmd buffer else NULL.
 * SQ or CQ definitions need to be used
 * to determine which ring of the ring pair this function should use. Typically SQ entry is retrieved by a requestor and CQ entry by the responder.
 * The intent when using this function is to fill the ring with data (it advances the head). rrr_ring_commit exposes data to the other side.
 */
void * rrr_ring_get_next_head_cmd_buf(struct rrr_ring_pair *ringp, bool isSubmit);

/*
 * Get next tail entry of command ring. This is for consuming data. Upon success it returns a pointer to the cmd buffer else NULL. Unlike
 * the get_next_head counterpart this function needs rdma_connection info, because it leverages RDMA (Read Op) to sync contents with remote
 * counterpart before returning the cmd data pointer. Typically responsder uses this function to retrieve new SQ entries and requestor uses it
 * to inspect CQ entries.
 */
void * rrr_ring_get_next_tail_cmd_buf(struct rrr_ring_pair *ringp, struct rdma_connection *res, uint32_t *idx, bool isSubmit);

/*
 *  Getting next tail fixed data buffer, both sides of the rings.
 */
void * rrr_ring_get_next_tail_fixed_data_buf(struct rrr_ring_pair *ringp, struct rdma_connection *res, uint32_t *idx, bool isSubmit);

int rrr_async_rdma_remote_data_buf_list(struct rrr_ring_pair *ringp, struct rdma_connection *res, uint64_t *laddr, uint32_t *lkey, uint64_t *raddr,\
        uint32_t *rkey, uint32_t *len, int entries);

void rrr_ring_data_buf_cli_release(struct rrr_ring_pair *ringp, bool isSubmit);

/*
 *  This function is employed when remote data buffer addresses are not apriori-known (fixed) but have arrived as part of the cmd. In this case,
 *  to enable aggregation this API allows to pull more than one remote buffers with single invocation.
 */
int rrr_ring_sync_burst_not_fixed_data_buf(struct rrr_ring_pair *ringp, struct rdma_connection *res,\
                                                uint32_t bufnum, uint64_t *rembfs, uint32_t* remkey, uint32_t * bsize, bool isSubmit);
/*
 * to be discussed if needed.
 */
int rrr_submit_ring_get_next_tail_free_data(struct rrr_ring_pair *ringp, uint32_t bufnum, uint64_t * localbfs, uint64_t *rembfs);

/*
 * This unlocks SQ or CQ rings for entry at idx. This allows the respective buffer to be recycled.
 */
void  rrr_ring_unlock_cmd_buf(struct rrr_ring_pair *ringp, uint32_t *idx, bool isSubmit);

/*
 * This unlocks data buffer for selected side and that allows them to be recycled.
 */
void  rrr_ring_synchronous_unlock_data_buf(struct rrr_ring_pair *ringp, uint32_t *idx, bool isSubmit);

/*
 * registers buffers post-initialization and pairs them with specific ring entries.
 */
int32_t rrr_register_buffer_for_transfer(struct rrr_ring_pair *ringp, struct rdma_connection *res,\
                                        unsigned char * buffer, uint32_t bfidx, uint64_t size);
/*
 * returns next SQ or CQ data ring head
 */
int rrr_ring_data_get_next_head(struct rrr_ring_pair *ringp, bool isSubmit);

unsigned char * rrr_ring_data_async_pool_get_free_buffer(struct rrr_ring_pair *ringp, int *bufidx, bool isSubmit);

int rrr_ring_data_async_get_next_head(struct rrr_ring_pair *ringp, int bufidx, bool isSubmit);

int rrr_ring_fixed_data_async_buf_list_release(struct rrr_ring_pair *ringp, int *bufidlist, int lsize);

int rrr_ring_commit(struct rrr_ring_pair *ringp, struct rdma_connection *res, bool isSubmit);
```

Figure 42: Available C functions in the RDMA Remote Ring library (librrr).

Currently, librrr has been successfully integrated as a I/O backed for deployment of vAccel framework, replacing the standard TCP-based communication layer.

5.5.2 Decentralized & Distributed Data Manager

The decentralised and distributed data exchange mechanism is carried out by the open-sourced communication middleware Eclipse Zenoh [5] , Figure 43 presents a characterization of its layered architecture, and the main possible configuration decisions to take into consideration when designing a real-time data exchange scenario.

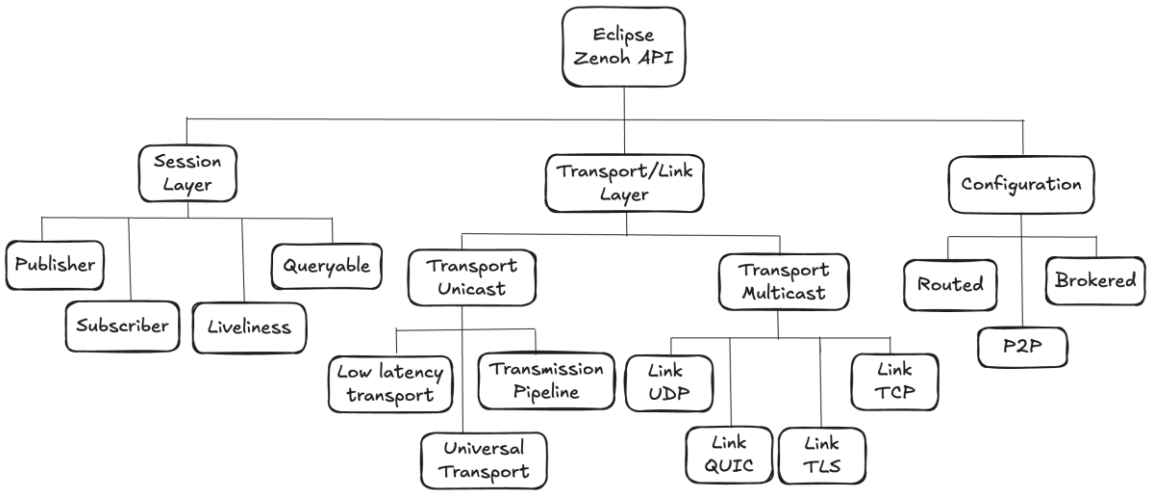


Figure 43: Characterization of the Eclipse Zenoh communication middleware.

Key components include:

- **The session layer** is the main entry point for an application to interact with Eclipse Zenoh. The developer should select what communication pattern is going to be used: pub/sub, queryable, or/and computations. The session establishment process supports dynamic discovery and is automatic between two Zenoh instances/processes running in the same network. The liveliness component is for monitoring system components (i.e., stay alive signal).
- **The Transport/Link layer** handles the establishment and management of transport connections, it also manages system's resources, and implements various transport protocols (TCP, TLS, UDP, QUIC, WebSocket, etc.).
- **The Configuration layer** is the central component for message routing and dispatching. The application developer can select to use either brokered, routed, or peer-to-peer communication patterns. Also, there is an admin space where the user can provide system monitoring and management capabilities. Zenoh also supports extensibility through dynamically loaded plugins that should be considered in the configuration phase.

5.5.3 Edge Storage Service

The Edge Storage Service (ESS) provides a simple-to-use, S3-compatible object storage solution to EMPYREAN applications. It has several unique features, including the ability to work with Associations, distributing erasure-coded data across cloud and edge storage locations. Users can establish exactly how and where their data resides by defining a storage policy and attaching it to an S3 bucket. Erasure coding, compression, and encryption parameters can also be configured. Many of these features rely on the SkyFlok.com backend, built to run CC's SaaS service with the same name. Among the many enhancements developed for EMPYREAN, the ESS introduces additional privacy-enhancing features such as the ability to keep encryption keys solely at the edge, inside the Association. It also provides a temporary autonomous function, making access to local storage possible, even when the link to the cloud is severed.

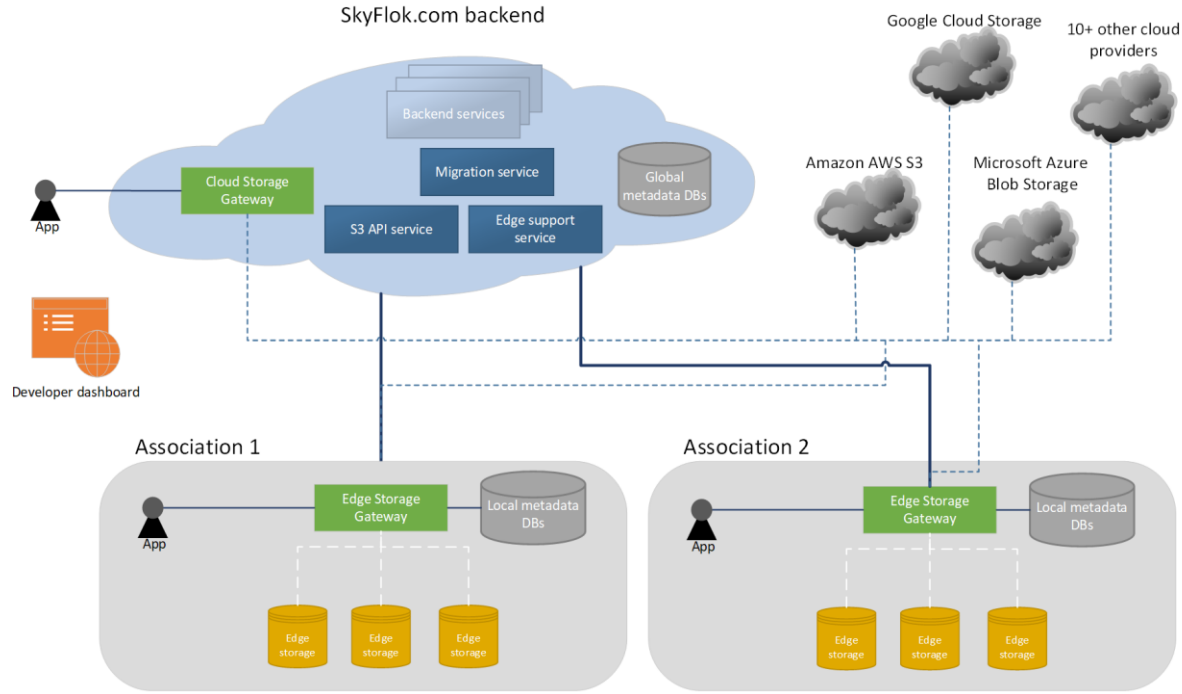


Figure 44: Overview of Edge Storage Service components.

The main components of the ESS are shown in Figure 44. Each Association has one Edge Storage Gateway that is deployed locally and provides access to the ESS' different public APIs. Storage resources are integrated using Edge storage devices, and a developer dashboard helps application developers set up their storage account. A Cloud Storage Gateway is also made available to users who are outside an association's network. However, this provides a limited feature set, in line with the association's security model, only providing access to data fragments that reside on public cloud providers. A detailed description of the service's features and components was included in Deliverable 3.1 (M15).

Custom CI/CD using Bitbucket Pipelines

The components of the ESS are being developed by Chocolate Cloud in-house, using the company's established CI/CD practices.

Figure 45: Example CI/CD pipeline run summary for one of the SkyFlok.com backend services.

To ensure code quality and adherence to project requirements, we have created a comprehensive set of rules and practices that guide us during component development. Here we provide a few examples:

- Each component must have unit tests that cover a large proportion of the code base. These unit tests are run automatically for each commit using a BitBucket pipeline, as shown in Figure 45.
- When developing a new feature, a separate git branch must be created. Before merging the branch, a pull request is created with two reviewers. Merging requires the approval of both reviewers and code coverage (see Figure 46) must not drop after the merge. Some critical components require additional checks by the lead software engineer.
- Issues are tracked using Jira and assigned to component owners. Priority is given to bugs.

To validate features in an end-to-end manner, we have created a separate S3 feature coverage test suite with roughly 130 test cases.

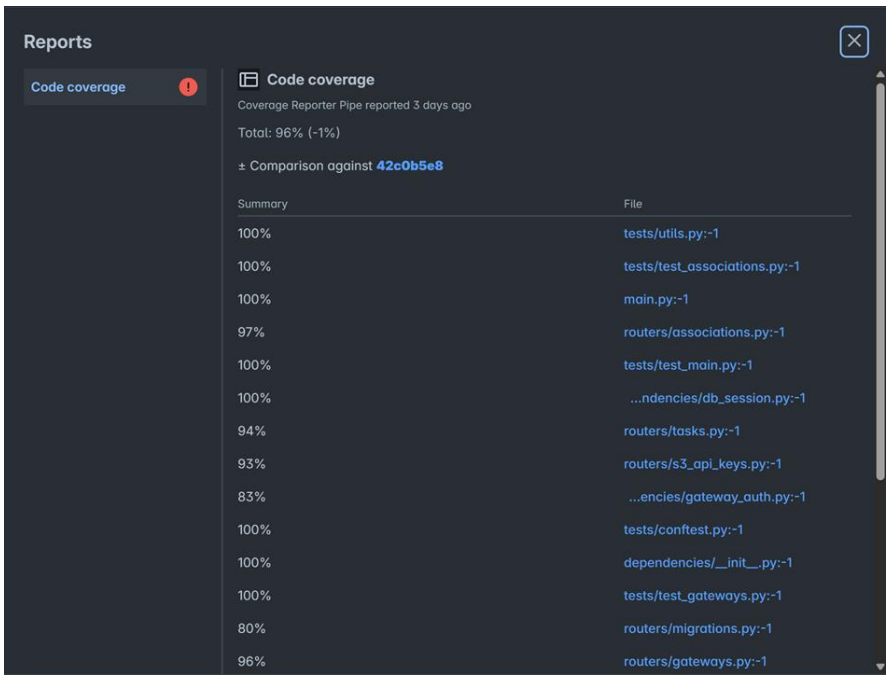


Figure 46: Output of BitBucket pipeline report on code coverage.

S3 feature coverage tests

We have created a test suite of roughly 130 individual tests that cover all S3 endpoints the ESS supports. In designing the suite, care was taken to cover different endpoint parameters and their combinations. Whenever the AWS S3 documentation was not clear on a certain aspect, functioning was validated using the AWS S3 cloud service.

At a high level, these tests have a two-fold role:

- Validate correct end-to-end functioning of S3 features.
- Check that ESS behaves in a compatible way with the AWS S3 cloud service.

Some tests check simple scenarios such as uploading and downloading objects. Others are focused on complex processes such as creating objects as part of a multipart upload. Yet others check interactions between different S3 endpoints, such as whether an S3 bucket with a started multipart upload can be deleted. Returned data types, headers, error codes, and messages are verified.

Integration and interfaces

S3-compatible API

The ESS S3-compatible object storage API supports all major Create, Read, Update, Delete (CRUD) features of both objects and buckets in their simplest form. Furthermore, support for multipart uploads is also present, along with the ability to perform ranged GetObject queries.

Buckets

- CreateBucket
- DeleteBucket
- ListBuckets

Objects

- GetObject
- HeadObject
- PutObject
- ListObjects
- ListObjectsV2
- ListObjectVersions
- DeleteObject

Multipart uploads

- CreateMultipartUpload
- AbortMultipartUpload
- CompleteMultipartUpload
- UploadPart
- ListMultipartUploads
- ListParts

An API reference can be found on Amazon's website [8]. Some notes regarding compatibility with AWS S3:

- From AWS's two addressing schemas, the ESS only supports *path-style* as there is no need to provide large-scale, DNS-based global routing of requests.
- Only versioned buckets are supported. We plan to add support to for non-versioned buckets.
- Authentication and request signing are performed using AWS Signature V4. Earlier methods are not supported.
- Less common transfer-encoding methods that chunk data are also supported.

Storage Policy API

The Storage Policy API will allow EMPYREAN users to create and retrieve storage policies programmatically. These can be thought of as recipes used to translate an application's storage requirements into a storage resource allocation. This API will be established shortly.

Storage Resource Telemetry API

The ESS will provide a list of supported Cloud Storage locations, along with their static characteristics. Beyond this, the performance of cloud locations is continuously monitored by the SkyFlok.com backend. The ESS will also expose this information through its Telemetry API. This API will be established shortly.

The list of Edge Storage devices is maintained by both the ESS and the EMPYREAN Registry. Their dynamic characteristics can be collected directly through a Prometheus-compatible interface exposed by MinIO [9]. No communication with the ESS is needed. This interface provides a way to both monitor performance characteristics as well as configure alert rules for certain types of abnormal events.

5.5.4 IoT Query Engine

The EMPYREAN platform introduces an "Analytics-Friendly Distributed Storage" solution specifically designed for IoT time-series data, building upon existing object storage with novel features. This system aims to balance the cost-effectiveness and reliability of erasure coding with the efficiency needed for querying time-series data. Unlike conventional systems that must scan entire objects or incur high costs by replicating data for analytics, EMPYREAN utilizes a new data alignment and coding scheme, incorporating Random Linear Network Coding (RLNC), to enable byte-level access without full file reconstruction. This approach significantly reduces data retrieval overhead while maintaining the benefits of erasure coding.

A key innovation being explored is the ability to compress data while retaining byte-level access, using Generalized Data Deduplication (GDD). This technique offers a deterministic relationship between uncompressed and compressed data, potentially further enhancing cost-effectiveness when combined with erasure coding. The project, a basic research effort by CC, has a low Technology Readiness Level (TRL) and focuses on establishing the feasibility and initial evaluation of these techniques. It addresses critical technical KPIs related to limiting data transfer costs and ensuring linear scaling of erasure-coded data retrieval for queries, directly contributing to EMPYREAN's objective of efficient data handling in cloud environments.

The IoT Query Engine will be integrated into the EMPYREAN platform in time for the next platform release.

5.6 Security, Trust, and Privacy Manager

5.6.1 p-ABC Library

The Privacy and Security Manager (PSM) will integrate a privacy-preserving Attribute-Based Credential (p-ABC) library as a foundational component across all its security functions within EMPYREAN. The p-ABC library enables the issuance, management, and verification of credentials that support selective disclosure of identity attributes, enhancing privacy while maintaining strong trust guarantees.

This library is critical for several core PSM capabilities:

- **Verifiable Credential Generation**: Allows the issuance of VCs where attributes are cryptographically bound but can be selectively disclosed by the holder.

- **Decentralized Identifiers (DIDs)**: Facilitate the generation of cryptographic keys and identifiers, especially for constrained devices and low-power IoT entities, enabling them to hold verifiable identities.
- **Attestation for Devices and Services**: Provides lightweight cryptographic proofs that can be used in attestation processes, ensuring device integrity and trustworthiness even in resource-constrained environments.
- **Privacy-Preserving Authorization**: Ensures that during access control and authorization workflows, only the minimum necessary information is shared, adhering to data minimization principles.

5.6.2 Privacy and Security Manager

The Privacy and Security Manager (PSM) is a core component of the EMPYREAN platform, enabling decentralized identity, verifiable credentials, secure authentication, and policy-based access control across Associations. Building upon Hyperledger Aries[3] and Fabric[4], and integrating OP-TEE for trusted execution, the PSM allows privacy-preserving identity flows and blockchain-backed authorization mechanisms. This section outlines the implementation progress of the PSM, describing key APIs, architectural features, and new capabilities.

Integration and interfaces

This section details the implementation progress of the PSM APIs and their integration into the broader EMPYREAN platform. It exposes a comprehensive set of RESTful interfaces that support key security and identity functionalities, allowing other platform components, such as the Aggregator and external services, to interact securely and consistently. The following subsections describe the different modules that compose the PSM's external interface.

We begin with Identity Management (Figure 47), which covers decentralized identifier (DID) creation and retrieval of trusted issuer registries. Then, we present the Credential Issuance interface (Figure 48), which enables enrolment, credential generation, and the construction and verification of Verifiable Presentations based on privacy-preserving attribute-based credentials. Following this, the JWT Signature module is explained, which provides mechanisms for signing and verifying JSON Web Tokens, including support for nested tokens used in chained authorization flows (Figure 49).

Next, the section covers TEE Management, which outlines the API for generating secure key pairs within the Trusted Execution Environment (OP-TEE) (Figure 50). Finally, we introduce the new Securing Resources capability, which allows EMPYREAN Aggregators to dynamically protect services by deploying Policy Decision Point (PDP)/Policy Enforcement Point (PEP) proxies, defining policy-based access requirements, and enforcing authorization decisions backed by blockchain and smart contracts (Figure 51).

[3] https://github.com/hyperledger/aries
[4] https://github.com/hyperledger/fabric

Verifiable Data Registry (Identity Manager)

These APIs handle the creation of Decentralized Identifiers (DIDs) and the retrieval of trusted issuers. They enable unique, verifiable identities across Associations and ensure credentials are issued only by authorized entities.

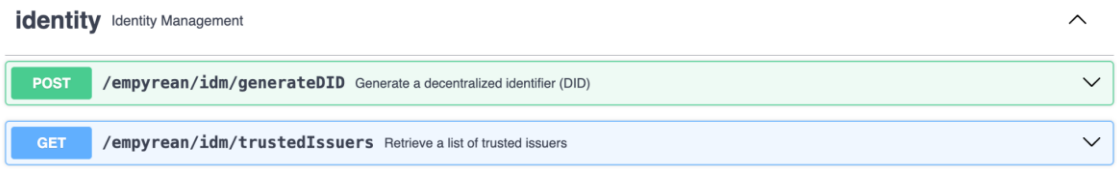


Figure 47: Privacy and Security Manager – Identity management REST API.

- **GenerateDID** (*POST /empyrean/idm/generateDID*): Generates a Decentralized Identifier (DID) for users or devices, enabling unique and verifiable digital identities.
- **GetTrustedIssuerList** (*GET /empyrean/idm/trustedIssuers*): Retrieves a list of pre-approved credential issuers stored in the Trusted Issuers Registry.

Credential Issuance

The Credential Issuance APIs support the enrolment of entities and the lifecycle of Verifiable Credentials (VCs). These calls allow users to receive credentials based on verified attributes, generate Verifiable Presentations (VPs) for selective disclosure, and verify the authenticity of submitted credentials.

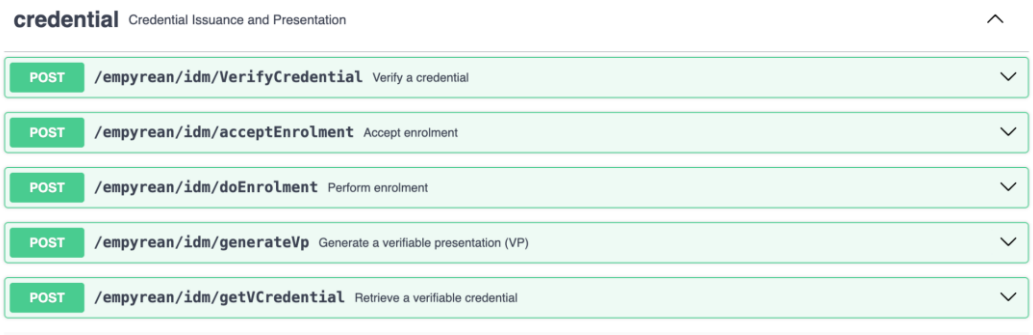


Figure 48: Privacy and Security Manager – Credential issuance REST API.

- **DoEnrolment** (*POST /empyrean/idm/doEnrolment*): Registers a new entity into the platform by issuing a Verifiable Credential based on verified attributes.
- **GetVCredential** (*POST /empyrean/idm/getVCredential*): Returns a previously issued VC associated with a DID.
- **GenerateVerifiablePresentation** (*POST /empyrean/idm/generateVP*): Constructs a Verifiable Presentation (VP) from one or more VCs, supporting selective disclosure via Zero-Knowledge Proofs.
- **VerifyCredential** (*POST /empyrean/idm/VerifyCredential*): Validates the authenticity, integrity, and issuer trust level of a presented VC or VP.

JWT Signature

These endpoints enable the signing and verification of JSON Web Tokens (JWTs), which are used to encapsulate identity claims and access rights. Support for nested JWTs allows for secure delegation and composability in authorization chains.

Figure 49: Privacy and Security Manager – JSW Signature REST API.

- **SignJWTContent** (*POST /empyrean/idm/signJWTContent*): Signs raw JSON payloads into JWTs, optionally embedding credentials or access rights.
- **SignJWTNested** (*POST /empyrean/idm/signJWTNested*): Signs a nested JWT that contains another signed JWT as payload. This is used to delegate access securely across chained authorization flows.
- **VerifyJWTContent** (*POST /empyrean/idm/verifyJWTContent*): Validates JWT signatures and checks expiration, issuer, and audience claims.

Secure & Trusted Execution Environment Management

This module provides a secure interface for generating key pairs within the Trusted Execution Environment (OP-TEE), ensuring hardware-isolated handling of sensitive cryptographic operations.

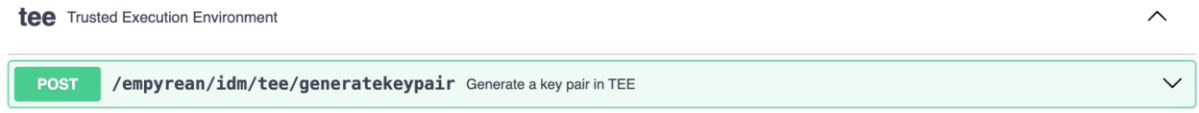


Figure 50: Privacy and Security Manager – Trusted Execution Environment (TEE) REST API.

- **TEE GenerateKeyPair** (*POST /empyrean/idm/tee/generatekeypair*): Generates a hardware-backed key pair using OP-TEE, isolating private key material from the host OS.

Securing Resources

This new feature allows the dynamic protection of services by registering resources and associating them with PDP/PEP proxies. The PSM enforces attribute-based policies and logs all access decisions on the blockchain for traceability.

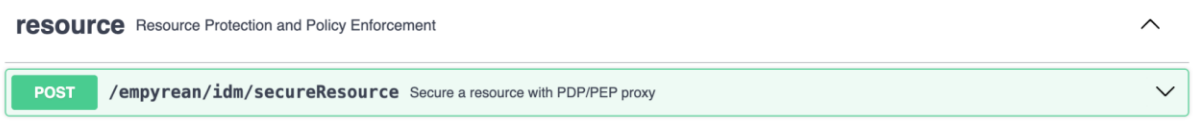


Figure 51: Privacy and Security Manager – Securing resources REST API.

- **SecureResource** (*POST /empyrean/idm/secureResource*) [New Feature – In Progress]:
  - This API allows Aggregators or EMPYREAN entities to register a service or resource (e.g., a URL, IP:port pair) that should be protected using dynamic authorization mechanisms.
  - Upon receiving this request, the PSM:
    - Deploys a dedicated PDP/PEP proxy component acting as a gateway to the protected resource.
    - Stores the corresponding access control policies as XACML rules via smart contracts on the blockchain.
    - Associates each policy with specific attribute-based access requirements (e.g., "role=technician" or "purpose=diagnostic").
  - The proxy returns a new port or secure endpoint through which the resource is now accessible.
  - Consumers must present a Verifiable Presentation (VP) and invoke **SignJWTContent** to obtain a valid token encoding their access claims. This token is then used to interact with the protected resource.
  - All access attempts are recorded on-chain using the access traceability smart contract, capturing:
    - The subject's DID
    - Token claims (attributes)
    - Resource identifier
    - Decision outcome (GRANTED/DENIED)
    - Timestamp

5.6.3 Cyber Threat Intelligence Engine

The EMPYREAN platform integrates a powerful Cyber Threat Intelligence (CTI) engine as a core component of its security, trust, and privacy framework. Designed to operate across the IoT-edge-cloud continuum, the CTI engine is responsible for automated cyber threat analysis, enabling the platform to identify, analyze, and respond to cybersecurity threats in a proactive and intelligent manner. By leveraging advanced mechanisms for threat detection, behavioral analysis, and intrusion response, the CTI engine empowers EMPYREAN to deliver robust and adaptive security capabilities tailored to complex, distributed environments.

One of the main features of the CTI engine is its ability to collect and analyze data from trusted external sources, including platforms such as the Cyber Threat Alliance (CTA) and the Malware Information Sharing Platform (MISP). By integrating with MISP, the CTI engine enables seamless ingestion, normalization, and correlation of cyber threat intelligence, providing

comprehensive situational awareness for both ongoing and emerging threats. The engine's architecture is designed to facilitate the periodic retrieval and processing of Indicators of Compromise (IoCs), structured threat data (e.g., STIX format), and other critical security insights that can be leveraged for real-time defense.

The CTI engine is also closely integrated with the EMPYREAN Telemetry Service, enabling it to ingest real-time monitoring data from across the platform. This tight coupling allows the CTI engine to provide context-aware threat intelligence, supporting automated security workflows, localized detection, and timely mitigation actions at both the Association and infrastructure levels. In addition, the CTI engine provides data and analytics interfaces that can be used by other EMPYREAN components, as well as by external security tools, ensuring compatibility with the broader cybersecurity ecosystem and facilitating the integration of intelligence-driven defense strategies into platform operations.

Through these capabilities, the EMPYREAN CTI engine forms the backbone of the platform's security posture—enabling proactive adaptation, rapid threat response, and continuous situational awareness across hyper-distributed IoT, edge, and cloud environments.

The API provided by the EMPYREAN CTI engine is purpose-built to empower both automated dashboards and human security experts with timely and actionable cyber threat intelligence. Recognizing the complexity and volume of security-relevant data within hyper-distributed environments, the API exposes structured endpoints that allow the platform's dashboards to visually present up-to-date CTI information, trends, and alerts in an accessible and intuitive manner.

This API-driven approach is essential for enabling situational awareness and decision support for security professionals operating the EMPYREAN platform. Through clearly defined RESTful endpoints, dashboards can retrieve detailed threat intelligence, query contextual information, visualize time series of cyber events, and analyze the ranking of malware families or specific incidents detected in the system. The design ensures that both real-time data and historical trends can be efficiently accessed and correlated.

A key architectural feature of the CTI engine is its seamless integration with the MISP API for the collection and normalization of cyber threat intelligence relevant to EMPYREAN. By leveraging the MISP API, the component can automatically ingest and process Indicators of Compromise (IoCs) and other structured threat data, ensuring the EMPYREAN CTI engine maintains a comprehensive and current view of the threat landscape. The collected intelligence is then made available to dashboards and experts through the EMPYREAN API, supporting a full cycle of intelligence-driven security operations—from data collection to actionable insight.

This design philosophy not only supports automation and proactive threat management via dashboards but also provides security experts with the tools they need to explore, understand, and respond to security events across the platform, leveraging both external intelligence (via MISP) and internally generated telemetry.

Figure 52: The CTI Engine REST API.

The EMPYREAN CTI engine exposes a suite of RESTful API endpoints, see Figure 52, designed to deliver actionable and contextual threat intelligence to both automated dashboards and human security analysts. Each endpoint is tailored to support a specific type of data retrieval or analysis relevant to the cybersecurity operations of the platform. Below, we describe the purpose and typical usage of each endpoint:

`/search`

The **/search** endpoint allows users and dashboards to query the CTI engine for objects or entities that match a given search term. This is typically used to quickly locate indicators, assets, or other threat intelligence objects within the EMPYREAN dataset, supporting investigative workflows and contextual analysis.

`/context`

The **/context** endpoint provides detailed contextual information about a specified term or indicator. When security experts or automated systems need to understand the background, related events, or threat associations of a specific entity, this endpoint returns enriched context drawn from both internal analysis and external intelligence sources.

`/timeSeries`

The **/timeSeries** endpoint is designed to deliver time series data for a given term or indicator. This supports trend analysis, anomaly detection, and temporal correlation of cyber events, enabling dashboards to visualize how threat-related activity evolves over time and allowing experts to track the progression or resolution of specific incidents.

`/rankingmalware`

The **/rankingmalware** endpoint offers insights into the prevalence or significance of different malware samples or families within the EMPYREAN platform. By retrieving rankings—either by individual malware or by family—security professionals can prioritize attention and resources toward the most impactful threats affecting their environment.

`/specialCases`

The **/specialCases** endpoint surfaces information on noteworthy or exceptional threat scenarios detected within the platform. This may include rare, novel, or otherwise significant events that warrant special attention, supporting proactive defense and targeted investigation.

`/bundleFigure`

The **/bundleFigure** endpoint returns detailed information for the creation of figures or visual representations associated with a specific bundle or incident identifier. This supports the visualization needs of dashboards and analytical tools, allowing experts to quickly interpret complex data and relationships within specific threat events.

These endpoints offer a comprehensive interface to the EMPYREAN CTI engine, enabling both automated systems and analysts to efficiently access, interpret, and act upon the rich threat intelligence collected and processed by the platform. By organizing the API in this modular fashion, the CTI engine facilitates both operational security monitoring and in-depth forensic analysis, aligning with EMPYREAN's mission of adaptive, intelligence-driven defence.

5.7 Monitoring and Observability Layer

5.7.1 Telemetry Service

The EMPYREAN Telemetry Service is a foundational component designed to provide real-time observability and situational awareness across distributed, device-rich Associations operating in the IoT-edge-cloud continuum. Tailored for heterogeneous infrastructures composed of embedded systems, IoT devices, edge clusters, and cloud services, the service enables intelligent, context-aware management by continuously collecting, aggregating, and exposing performance and status data.

At the heart of the telemetry infrastructure lies a modular and distributed architecture, where multiple Telemetry Agents are deployed close to the data sources, either on IoT nodes, edge platforms, or orchestrated containers. These agents handle local data collection and preliminary filtering before forwarding selected metrics to Telemetry Aggregators at the Association level. Each Aggregator is responsible for correlating and normalizing telemetry from different local platforms, enabling high-level visibility across the Association.

The system supports device registration and association mapping, automatically linking telemetry data to specific devices, applications, and orchestration workflows. This association-aware telemetry model ensures traceability and enables targeted monitoring and diagnostics in scenarios involving large fleets of constrained or mobile IoT devices.

Data is exposed through standardized interfaces (e.g., REST, gRPC, Prometheus APIs) to platform components such as the Orchestration Manager, Analytics Engine, and CTI Engine, enabling closed-loop automation, threat detection, and optimization. The telemetry service is tightly integrated with the orchestration stack, adapting its data collection strategies to events such as device onboarding, service deployment, scaling, and failure recovery. Leveraging extensible backends like Prometheus, it supports persistent storage and time-series analysis for real-time and historical insight.

Designed to operate under resource and connectivity constraints typical of IoT environments, the EMPYREAN Telemetry Service incorporates lightweight components and edge-level processing to ensure scalability, resilience, and low-latency observability in hyper-distributed

deployments. Its extensible design allows seamless integration of new device types and metrics, making it a versatile telemetry backbone for intelligent and autonomous operations across EMPYREAN Associations.

Figure 53 presents a high-level summary of the Telemetry Service deployment, illustrating the end-to-end flow of telemetry data, from collection to consumption across the platform.

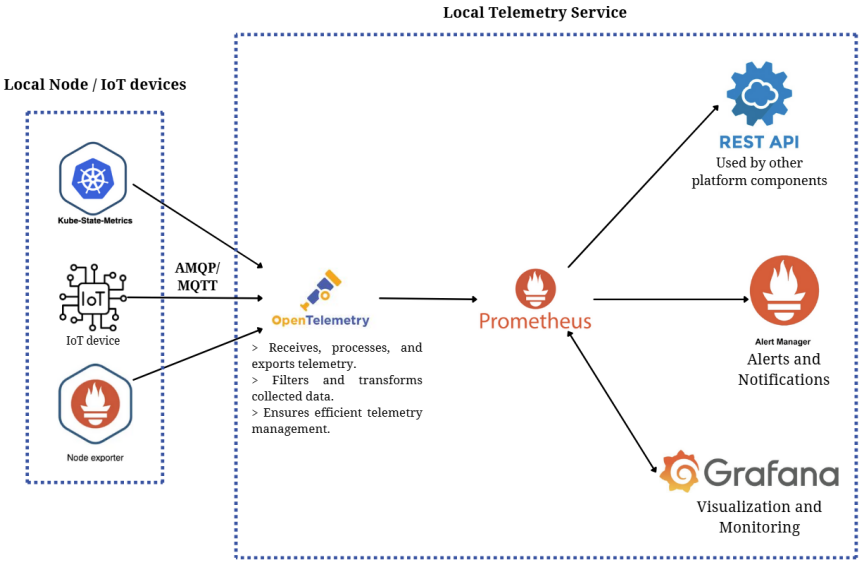


Figure 53: EMPYREAN Telemetry Service deployment.

Integration and interfaces

To support dynamic observability in distributed environments, the service exposes well-defined interfaces—both internal and public—that allow efficient interaction with devices, agents, orchestration components, and higher-level analytics modules. These interfaces are designed to ensure smooth data flow across the telemetry pipeline, from collection at the edge to processing, storage, and consumption by platform services and external users.

Integration with IoT Devices and Edge Environments

One of the core design priorities of the telemetry infrastructure is to support telemetry data ingestion from diverse IoT devices, often operating under strict constraints in power, bandwidth, or computational resources. For this purpose, the EMPYREAN Telemetry Service includes support for lightweight telemetry agents deployed on devices. These agents can collect local metrics, such as CPU load, temperature, memory usage, sensor readings, and network status, and forward them via lightweight messaging protocols like MQTT or AMQP.

To ensure seamless ingestion of such device-originated telemetry, specialized receivers within the OpenTelemetry (OTEL) Collector are configured to handle these protocols and data formats. This allows raw or pre-processed telemetry data to be received, normalized, and forwarded for further processing, regardless of the device's hardware or operating system. The system also enables association-aware telemetry, tagging incoming metrics with

identifiers that bind the data to specific nodes, applications, or user-defined associations, facilitating fine-grained analysis and traceability.

Internal Interfaces and Pipeline Management

The telemetry infrastructure supports a robust set of internal interfaces that govern the configuration, management, and orchestration of telemetry pipelines. These interfaces are exposed through a dedicated agent API, built with FastAPI, which enables the dynamic management of telemetry pipelines at runtime.

Key functionalities of the agent API include:

- **Dynamic pipeline creation and modification**: APIs support the injection or removal of OpenTelemetry pipeline components (e.g., receivers, processors, exporters) through declarative updates to the OTEL Collector configuration.
- **Lifecycle operations**: Endpoints allow operators and automated systems to start, stop, restart, or reconfigure telemetry pipelines without service interruption.
- **Namespace-aware integration**: The API operates across Kubernetes namespaces, applying configuration changes via ConfigMaps and monitoring pods using label selectors and controller logic.
- **Component inspection**: Clients can query current pipeline status, list telemetry agents, and perform health checks across the telemetry mesh.
- **Reload operations**: Secure endpoints allow on-demand configuration reloads, minimizing disruption during updates or deployments.

This internal control layer ensures that the telemetry service can adapt in real time to infrastructure changes, such as the onboarding of new devices, service migrations, or scaling actions initiated by the orchestration layer.

5.7.1.1 Data Access via Public APIs

To enable data access via public APIs and support data consumption by platform services and users, the service exposes telemetry metrics through two primary public interfaces: the Prometheus API and the Agent API.

The Prometheus API provides access to real-time and historical time-series telemetry via PromQL endpoints. It includes metrics such as CPU usage, memory consumption, disk I/O, energy usage, and custom application-specific data. These endpoints are utilized by internal modules like the Decision Engine, Analytics Engine, and CTI Engine, as well as by external tools like Grafana for dashboard generation. As illustrated in Figure 54, the API supports time-range filtering, label-based querying, and data aggregation, making it ideal for both manual debugging and automated analytics workflows. For instance, the Analytics Engine may use historical telemetry for anomaly detection, while the Decision Engine relies-on real-time usage trends to optimize workload scheduling across devices and Associations.

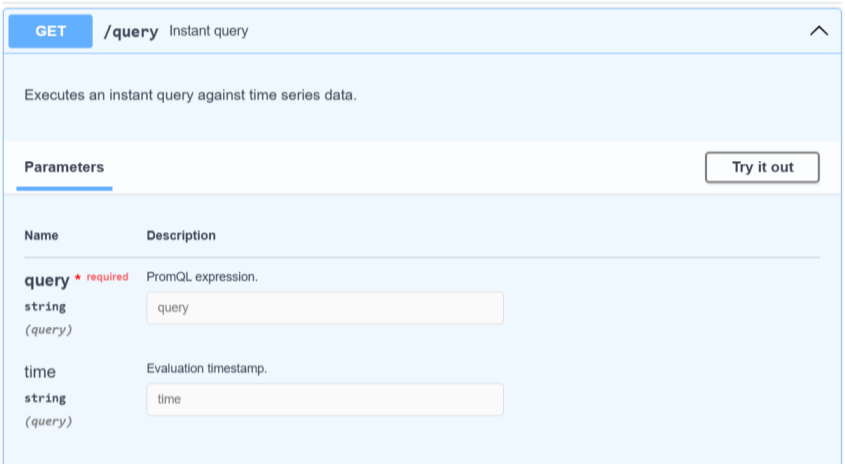


Figure 54: Telemetry Service Prometheus API.

The Agent API is a RESTful interface that enables interaction with the telemetry infrastructure. It allows developers and orchestration components to dynamically reconfigure pipelines, add or remove telemetry sources, and adapt metrics collection to evolving system conditions. The initial version of the API (Figure 55), includes several methods for listing, updating, and deleting pipelines, as well as reloading the configuration in real time.

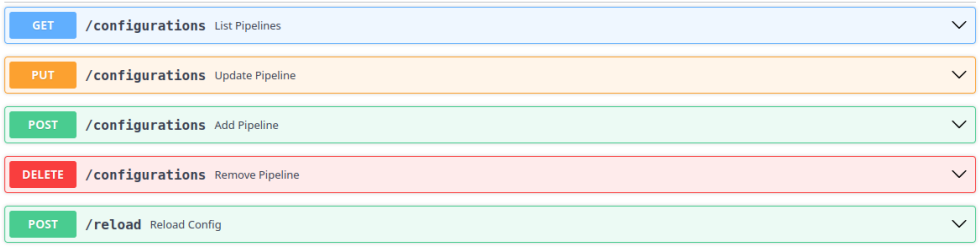


Figure 55: Telemetry Service Agent REST API.

These public APIs enable open integration with other EMPYREAN components, ensuring interoperability and extensibility in diverse environments.

Integration with EMPYREAN Services

The Telemetry Service plays a pivotal role in coordinating with other key components of the EMPYREAN architecture:

- **Decision Engine**: Consumes real-time metrics for performance-aware scheduling, load balancing, and energy-aware workload placement. Integration is achieved through Prometheus queries to retrieve the latest system state across Associations and devices.
- **Analytics Engine**: Leverages long-term telemetry data from Prometheus to train machine learning models and detect anomalies. Integration occurs through RESTful queries and subscription to telemetry topics.
- **Orchestration Engine**: Sends events (e.g., deployment, migration, scaling) to the telemetry infrastructure to trigger corresponding adjustments in monitoring

configurations. In return, it consumes telemetry outputs to make informed orchestration decisions.

- **CTI Engine**: Subscribes to telemetry streams related to system behavior and security events. Correlates monitoring signals with Indicators of Compromise (IoCs) or threat intelligence for early detection and mitigation.

This bidirectional communication ensures that the telemetry service is not only a passive observability tool but an active enabler of intelligent automation and proactive system adaptation.

Scalability and Federation Across Associations

In line with EMPYREAN's architecture, the Telemetry Service supports federated operation across multiple Associations. Each Association deploys one or more Telemetry Aggregators responsible for: (i) collecting and correlating metrics from all devices, applications, and agents within its scope, and (ii) propagating relevant metrics to global services for inter-Association coordination. This distributed model ensures scalability, fault isolation, and low-latency telemetry handling, while still providing global visibility and unified access to data.

Visualization Interfaces

Telemetry data is visualized via integrated Grafana dashboards, which connect to the Prometheus API. Dashboards are designed to support:

- Device-level monitoring and health status.
- Association-level resource usage and topology overviews.
- Real-time event tracking and incident diagnosis.
- Historical trend analysis for predictive maintenance.

5.7.2 The Analytics Engine

The Analytics Engine enables autonomous operation and adaptive self-management across the Association-based continuum. The EMPYREAN platform deploys multiple instances of the Analytics Engine, each leveraging real-time telemetry data to implement distributed service assurance mechanisms. These engines apply continuous analysis techniques to verify that applications operate as expected and proactively or reactively trigger re-optimization actions to maintain optimal performance, reliability, and efficiency across the platform.

The Analytics Engine is designed as a modular and scalable microservices-based system, comprising four core components: Access Interface, Data Connector, Data Manager, and Event Detection Engine. Figure 56 illustrates the engine's architecture, detailing its main components and their interactions with other EMPYREAN services. Additional details regarding the design, initial implementation, and exposed interfaces of the Analytics Engine are provided in D3.2 (M15).

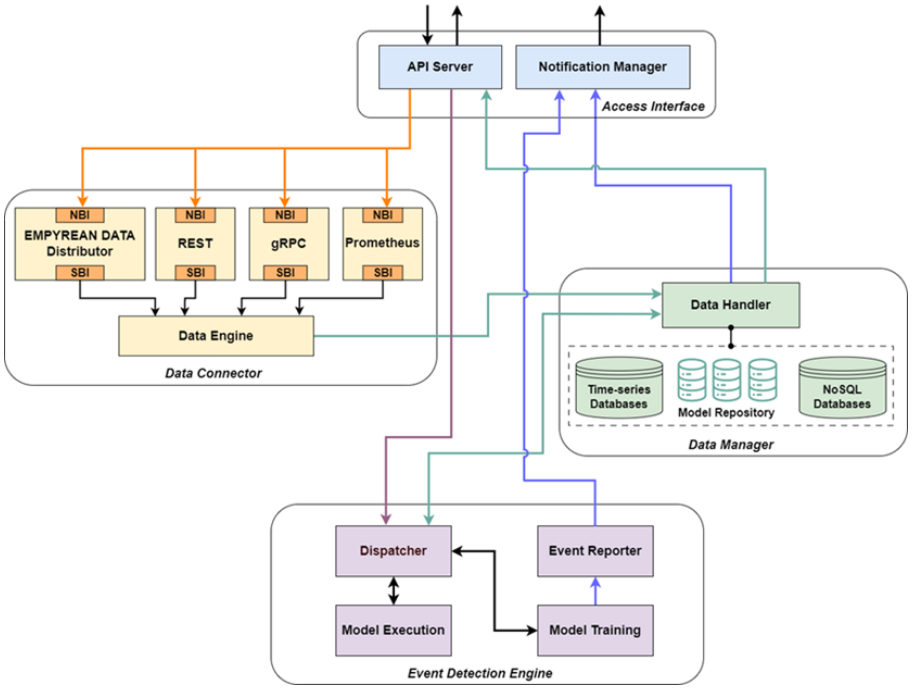


Figure 56: Analytics Engine architecture and core components.

Integration and interfaces

The initial version of the EMPYREAN Analytics Engine delivers the core functionalities of the Access Interface, Data Connector, and Data Manager services. Implemented in Python and adhering to a microservices architecture, each service exposes a well-defined set of interfaces that enable seamless interaction, both internally among Analytics Engine components and externally with other key platform services, such as the Telemetry Service, EMPYREAN Aggregator, and Service Orchestrator.

The Access Interface enables bidirectional communication to exchange commands, information, and notifications among the Analytic Engine instances and other services within the distributed EMPYREAN control and management plane. The Data Connector service manages the collection of raw monitoring and streaming telemetry data from various sources within the Monitoring and Observability layer. Finally, the Data Manager manages data storage and facilitates data exchange between internal and external components, providing local storage of processed data, trained models, and analysis results.

All services are developed, integrated, and packaged through the EMPYREAN CI/CD pipeline, ensuring consistent quality and streamlined deployment. Dedicated container images are provided for each Analytics Engine service, accompanied by their respective configuration files, which support automated deployment within the EMPYREAN testbed infrastructure.

Next, we summarize the initial set of implemented REST APIs and asynchronous communication interfaces exposed by the component services, see Figure 57, and Figure 58.

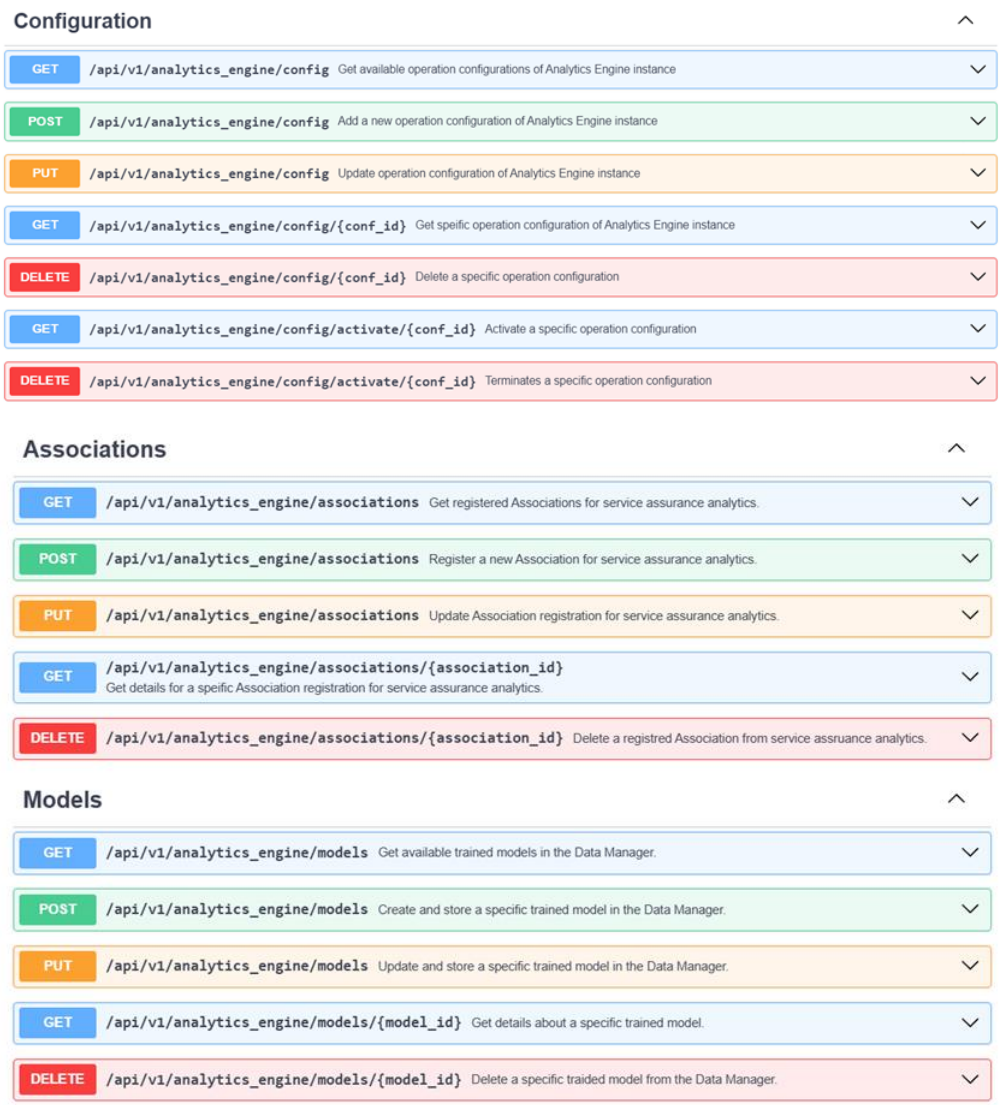


Figure 57: Analytics Engine – Access Interface RESTful API.

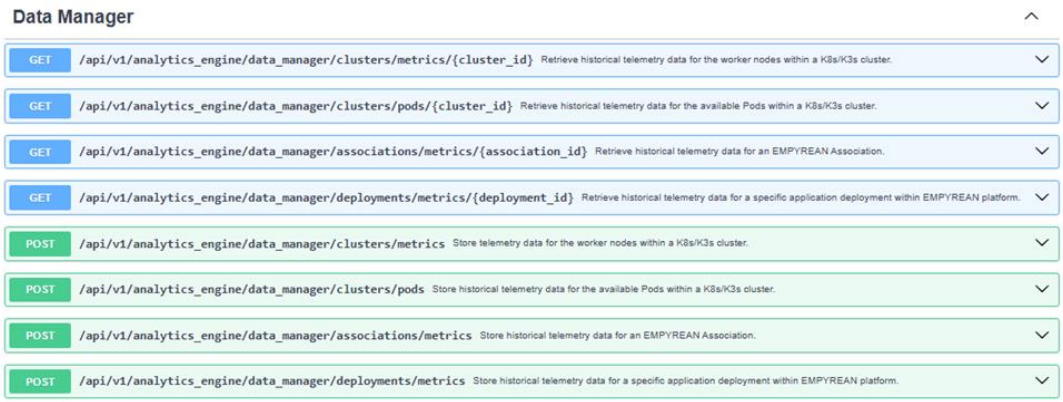


Figure 58: Analytics Engine – Data Manager RESTful API.

Analytics Engine and Associations

Upon the creation of a new Association within the EMPYREAN platform, the responsible EMPYREAN Aggregator performs several operations, including notifying the corresponding Analytics Engine about the newly defined Association. The complete operational flow for this process is detailed in Section 6.2. This interaction enables the Analytics Engine to seamlessly collect relevant data and perform service assurance activities at the Association level, ensuring both the correct operation of participating resources and the compliance of deployed services with the desired state.

To facilitate this, the Aggregator communicates with the Access Interface service of the Analytics Engine by invoking its exposed REST API. Specifically, it issues the following POST request to register the new Association

```
POST /api/v1/analytics_engine/associations

{"association_uuid": "ebe12f54-a9cb-476a-b930-4befb0236fce", "aggregator_uuid":
"a2b66fa3-3f13-416b-8644-56328e0de4ba","policy_uuid":"5c0eed16-31a8-467d-83db-
ef1010466755"}
```

Additionally, the following GET method can be used to retrieve a list of active Associations registered within a given Analytics Engine instance.

```
GET /api/v1/analytics_engine/associations

[{"association_uuid":"ebe12f54-a9cb-476a-b930-4befb0236fce", "aggregator_uuid":
"a2b66fa3-3f13-416b-8644-56328e0de4ba", "policy_uuid": "5c0eed16-31a8-467d-83db-
ef1010466755", "status": "active", "enabled_at": 1751639172}]
```

Data connectors registration and management

The implementation enables the dynamic registration and configuration of various Data Connectors via a common northbound REST interface, see Figure 59, exposed by the Data Connector service. During initialization, each Data Connector is expected to register with a specific Analytics Engine instance automatically.

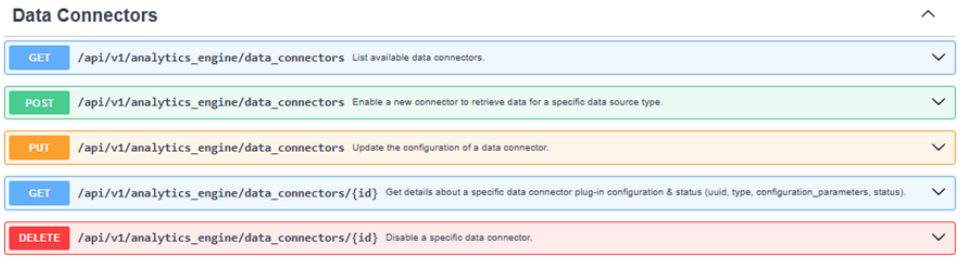


Figure 59: Analytics Engine – Data Connector plug-ins RESTful northbound interface.

This registration is executed by invoking the corresponding POST method. Below is an example showing how to register a Data Connector designed to collect information from the EMPYREAN Telemetry Service through the Prometheus interface.

```
POST /api/v1/analytics_engine/data_connectors

{"uuid": "ddced532-2c76-4557-9be1-2be622cbdcee", "connector_type":
"prometheus_connector", "mode": "pulling", "connection_parameters":
{"service_url":"http:147.102.16.114:90", "authentication":{"token": "ACCESS_TOKEN"}},
"operational_parameters": {"retain_period": 900, "retrieval_interval":180,
"caching_expiration_period": 7200, "active_notifications":true, "namespaces":
["empyrean", "empyrean_integration"]}}
```

In addition to registration, the operation parameters of a Data Connector can be dynamically updated using the exposed PUT method. In the following example, we modify the active configuration of a previously registered data Connector (UUID "ddced532-2c76-4557-9be1-2be622cbdcee") by setting the data pulling period to five minutes and disabling the emission of operational notifications.

```
PUT /api/v1/analytics_engine/data_connectors

{"uuid":"ddced532-2c76-4557-9be1-2be622cbdcee","operational_parameters":
{"retrieval_internal":300,"active_notifications": false}}
```

Data Manager

The Data Connectors use the appropriate POST methods exposed by the Data Manager to submit collected monitoring data. This interface enables the continuous ingestion of time-series metrics into the Analytics Engine. The following request demonstrates the update of collected data related to deployed services within a specific Association.

```
POST /api/v1/analytics_engine/data_manager/associations/metrics

{"association_uuid":" ebe12f54-a9cb-476a-b930-
4befb0236fce","metrics":[{"cluster_uuid":"7628b895-3a91-4f0c-b0b7-
033eab309891","creation_timestamp":1751701148,"deployment_uuid":"fdf45855-1299-47f1-
8ea6-98be8d89030b","name":"data-producer-475b4c6b85-k89b5","namespace":"empyrean-
integration","node":"wn2-
serifos","phase":"Running","restarts":0,"start_time":1751701150,"timestamp":1751701848
,"usage":{"cpu":"1002310835n","memory":"150672Ki"}},{"cluster_uuid":"7628b895-3a91-
4f0c-b0b7-033eab309891","creation_timestamp":1751701148,"deployment_uuid":"fdf45855-
1299-47f1-8ea6-98be8d89030b","name":"state-manager-5b4c6b4785-
g7jb5","namespace":"empyrean-integration","node":"wn2-
serifos","phase":"Running","restarts":0,"start_time":1751701150,"timestamp":1751701848
,"usage":{"cpu":"1006083037n","memory":"175124Ki"}},{"cluster_uuid":"7628b895-3a91-
4f0c-b0b7-033eab309891","creation_timestamp":1751701148,"deployment_uuid":"fdf45855-
1299-47f1-8ea6-98be8d89030b","group_id":"wn4-santorini","name":"classifier-21a9d90c6a-
5b4c6","namespace":"empyrean-integration","node":"wn4-
santorini","phase":"Running","restarts":0,"start_time":1751701151,"timestamp":17517018
49,"usage":{"cpu":"100476936n","memory":"204008Ki"}}]}
```

The Event Detection Engine can access the collected monitoring and historical data through the GET methods exposed by the Data Manager service. This REST interface enables the retrieval of time-series or contextual information necessary for event analysis and anomaly detection. For instance, the following request retrieves the available monitoring data for a specific worker node of a Kubernetes cluster within the EMPYREAN platform (ICCS K8s with UUID 7628b895-3a91-4f0c-b0b7-033eab309891).

```
GET /api/v1/analytics_engine/data_manager/clusters/metrics/7628b895-3a91-4f0c-b0b7-
033eab309891?wn=wn2-serifos
```


```
{"cluster_uuid":"7628b895-3a91-4f0c-b0b7-
033eab309891","timestamp":1751703648,"nodes":[{"node_cpus":[{"idle":14649163.75,"label
":"0","used":1011547.1},{"idle":14403219.21,"label":"1","used":1245255.79},{"idle":142
73190.45,"label":"2","used":1378752.47},{"idle":14093762.62,"label":"3","used":1559953
.59}],"node_filesystem_avail_bytes":60301340672,"node_filesystem_free_bytes":603181178
88,"node_filesystem_size_bytes":207929917440,"node_filesystem_usage_percentage":70.99,
"node_filesystem_used_bytes":147611799552,"node_memory_Buffers_bytes":1945653248,"node
_memory_Cached_bytes":57360580608,"node_memory_MemAvailable_bytes":64005013504,"node_m
emory_MemFree_bytes":1168969728,"node_memory_MemTotal_bytes":67434598400,"node_memory_
MemUsed_bytes":66265628672,"node_memory_usage_percentage":98.27,"node_name":"wn2-
serifos","node_total_running_pods":14}]}
```

6 Platform Integration and Operation Flows

This section demonstrates how the components developed and integrated as part of the initial EMPYREAN release support the key operation flows defined in Deliverable D2.3 (M12). By showcasing end-to-end interactions among the core platform services, we illustrate the orchestration, coordination, and automation capabilities of the EMPYREAN platform across the IoT-edge-cloud continuum. These flows validate the implementation and integration work carried out, highlighting the platform's readiness to support secure, dynamic, distributed, and intelligent management of an Association-based continuum.

6.1 Entity Enrolment, Security, and Resource Protection

The EMPYREAN platform integrates a comprehensive framework for resource protection, enrolment, and access governance, establishing a robust security mechanism that secures critical platform components from the first stages of operation. This framework is based on the coordinated interaction between the Privacy and Security Manager (PSM), the EMPYREAN Aggregator, and the policy enforcement infrastructure composed of the Policy Enforcement Point (PEP), the Policy Decision Point (PDP), and an immutable traceability layer powered by Distributed Ledger Technology (DLT).

In the EMPYREAN architecture, the term resource broadly refers to any protected asset exposed within the platform, including services, APIs, data endpoints, or computational functions. The first critical resource protected by this mechanism is the EMPYREAN Controller, as it manages the onboarding of resources across the platform. The EMPYREAN Aggregator initiates this protection process, ensuring that this central management service is fully secured before being exposing it to any enrolled entity. Although, the initial integration focuses on protecting the EMPYREAN Controller, the same mechanism applies universally to any resource registered within an Association.

6.1.1 Resource protection and access workflow

The complete protection and access workflow is illustrated in Figure 60, showing how a resource is protected, accessed by authorized entities, and how all actions are securely logged.

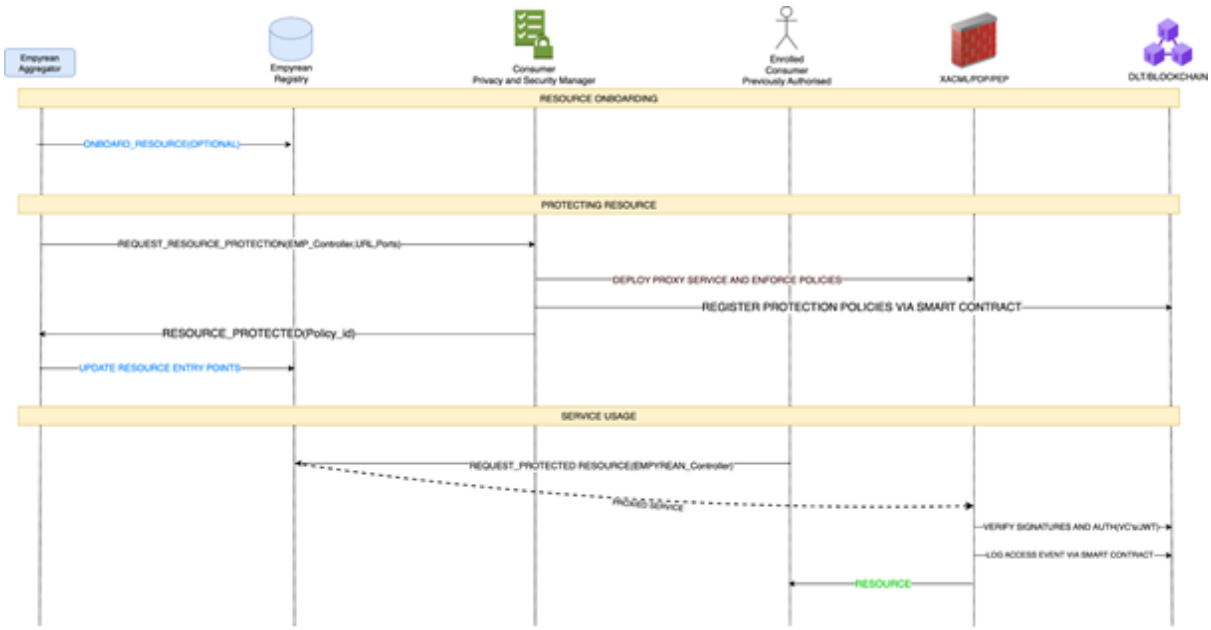


Figure 60: Workflow for protecting and accessing resources in EMPYREAN, starting with the Controller as the first protected resource. The Aggregator initiates protection, PSM enforces policies via PEP/PDP, and all access is verified and immutably logged on blockchain.

6.1.1.1 Workflow description

The process starts with the optional onboarding of the resource into the EMPYREAN Registry, where the Aggregator records its metadata, endpoints, and other relevant details. This ensures that the resource is recognized and discoverable within the platform.

The Aggregator then sends a Request Resource Protection to the PSM, specifying the resource identifier, network endpoints, and the access policies that define who can interact with the resource. The PSM processes this request by deploying a PEP proxy and enforcing the policies through the PDP, which dynamically evaluates every access request. The policies are also registered via a smart contract on the blockchain, ensuring traceability and integrity.

Once the protection is in place, the PSM returns a Policy ID to the Aggregator. This identifier not only confirms the protection status but also establishes ownership of the protection policy for that resource. The Aggregator can use this Policy ID to update later, refine, or revoke the policy, enabling the protection to evolve dynamically. For example, the policy can be updated to restrict access further, loosen constraints, or adapt to new operational or security requirements. The Aggregator also updates the EMPYREAN Registry with the secured resource's updated endpoints.

When an Enrolled Consumer, an entity properly onboarded and holding the necessary Verifiable Credentials (VCs) or JWTs derived from VCs, attempts to access the protected resource, the request is intercepted by the PEP. The PDP verifies the presented credentials, checking their signatures through the associated Decentralized Identifiers (DIDs) and evaluating their attributes against the enforced policies. If the access is authorized, the request is forwarded to the resource (e.g., the EMPYREAN Controller), which returns the appropriate response.

6.1.1.2 Generalization to all EMPYREAN resources

While the EMPYREAN Controller serves as the initial resource protected via this mechanism, the architecture is designed to generalize across all resources within the platform. Any service, API, data endpoint, or computational function can be secured following the same workflow, ensuring consistent and scalable security governance across the EMPYREAN ecosystem.

6.1.2 Secure device attestation and lightweight identity management

An integration strategy has been defined that combines secure device attestation, decentralized identities (DIDs), and Verifiable Credentials (VCs), specially tailored for constrained IoT devices. As illustrated in Figure 61, the approach leverages the Device Identifier Composition Engine (DICE) mechanism to derive cryptographic keys at boot time, tightly coupling the device's identity to the integrity of its firmware. The integration flow illustrates DICE-based key derivation, generation of Verifiable Credentials, and manufacturer validation using Decentralized Identifiers (DIDs) or X.509 certificates.

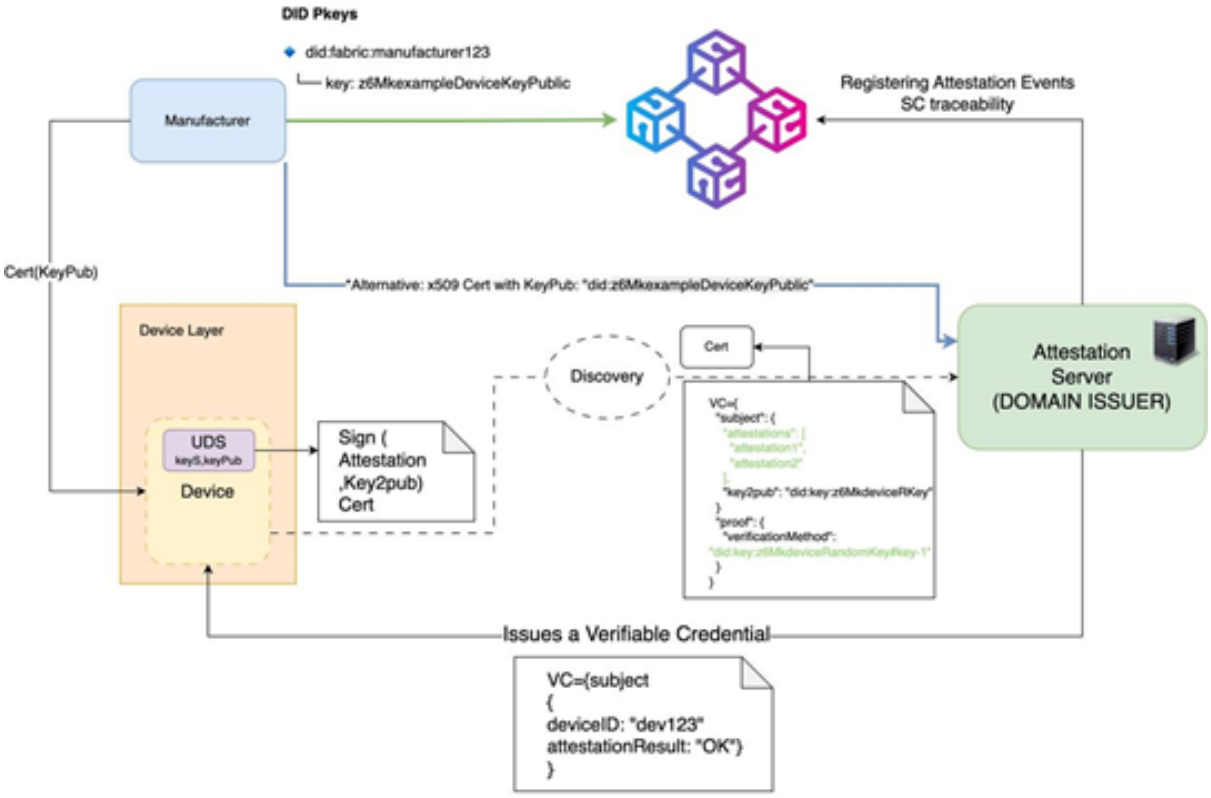


Figure 61: Secure attestation and decentralized identity integration flow involving Manufacturer, Device, Attestation Server, and Blockchain.

At manufacturing, the Manufacturer registers a decentralized identifier (e.g., did:fabric) on a blockchain or decentralized ledger, exposing public keys that correspond to specific devices or device families. As an alternative, the manufacturer may issue an X.509 certificate

embedding the did:fabric identifier in a custom field, enabling subsequent key validation in both decentralized and traditional public key infrastructure (PKI) environments.

When a device boots, it signs an attestation package that contains:

- Firmware measurements.
- A public key derived via DICE mechanism.
- A self-issued Verifiable Credential (VC) embedding these evidences.

The device forwards this information through a discovery mechanism (such as Akri) to the Attestation Server, which validates the attestation using the manufacturer's public keys (retrieved via the DID document or X.509 certificate). The Attestation Server may also:

- Register traceability events on the blockchain.
- Act as an issuer of domain Verifiable Credentials, confirming the device integrity (e.g., attestationResult: OK).

To accommodate the constraints of IoT resource-limited devices, the solution integrates the p-ABC library (Privacy-preserving Attribute-Based Credentials) instead of deploying a full Privacy and Security Manager (PSM). This allows devices to produce selective disclosure proofs and privacy-preserving verifications with minimal computational overhead.

6.1.3 Secure user access via Privacy and Security Manager and RYAX Workflow integration

The EMPYREAN platform supports privacy-preserving user authentication and authorization across the IoT-edge-cloud continuum by integrating its Privacy and Security Manager (PSM) with the RYAX Workflow Engine. Incorporating Keycloak[5] as the underlying Identity and Access Management (IAM) system, this integration (Figure 62) provides a unified and comprehensive mechanism for managing user identities, attributes, and controlled access to protected services.

This integration involved the coordinated operation of several core actors and components.

1. Privacy and Security Manager (PSM): Authentication and Credential Issuance

The PSM authenticates entities and issues privacy-preserving Attribute-Based Credentials (p-ABCs) in the form of Verifiable Credentials (VCs). These credentials securely encode user or platform entity attributes, enabling privacy-preserving authorization across services. At the core of this process is the p-ABC library, a critical cryptographic module that enables advanced privacy techniques such as Zero-Knowledge Proofs (ZKPs), selective disclosure, and unlinkable token generation. This allows entities to prove specific required attributes without exposing additional information, ensuring secure and private access across EMPYREAN's distributed services.

[5] https://www.keycloak.org

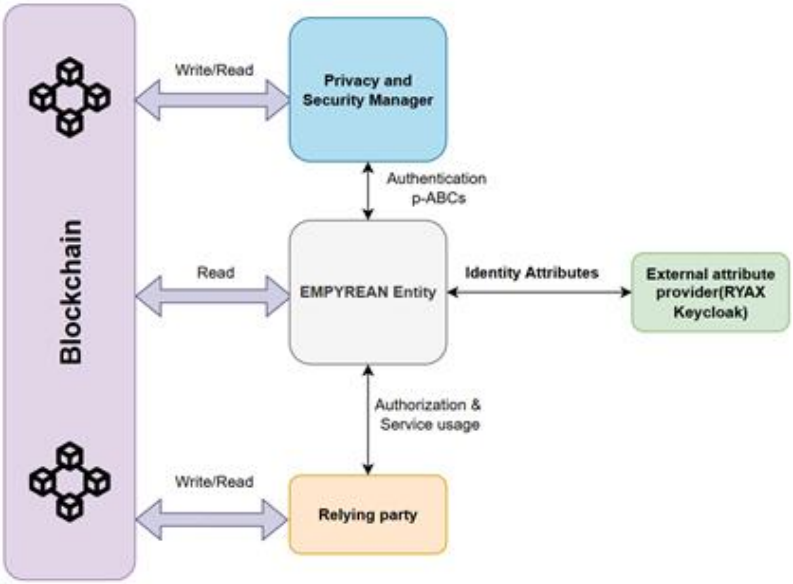


Figure 62: Workflow of the EMPYREAN Privacy and Security Manager integrating RYAX and Keycloak for attribute-based authentication and dynamic access control.

2. External Attribute Provider (RYAX Keycloak)

Keycloak, embedded within the RYAX platform, serves as the external attribute provider, responsible for certifying entity attributes such as roles, permissions, or organizational affiliations. These certified attributes are retrieved by the Relying Party from Keycloak and forwarded to the PSM as part of the p-ABCs issuance process. This procedure forms a bridge between EMPYREAN's decentralized, privacy-preserving credential ecosystem and established IAM systems.

3. EMPYREAN Entity

An EMPYREAN Entity represents any participant in the platform that enrolls into the system to access services. This can include users, devices, applications, or agents that interact with EMPYREAN's services. Once enrolled and issued the appropriate p-ABCs, the EMPYREAN Entity can interact with services protected by dynamic resource protection mechanisms, such as privacy-preserving proxies or other access control systems enforced by the PSM. These protections ensure that all access is mediated through policies based on attributes rather than static identities, enabling fine-grained, context-aware control.

4. Relying Party

The Relying Party is any platform component or service that requires verifying attributes and requesting credentials to access or enable access to protected services. It retrieves entity attributes from Keycloak and requests the issuance of p-ABCs from the PSM. Armed with these credentials, the Relying Party can then access services safeguarded by the platform's dynamic access control mechanisms (e.g., ABAC-enforced proxies), ensuring compliance with security and privacy policies.

5. Blockchain as a Verifiable Data Registry

The EMPYREAN architecture employs a Blockchain-based Verifiable Data Registry to securely anchor credential issuance, policy enforcement, and potential revocations. This registry ensures transparency, immutability, and auditability of the trust and access control processes, establishing a reliable foundation for secure interactions across decentralized associations.

This integration enables the following key functionalities:

- **Entity Enrolment**: Enrolment of EMPYREAN Entities (users, devices, applications) into the platform, enabling them to interact securely with services.
- **Attribute Certification via Keycloak**: Relying Parties obtain certified attributes from Keycloak to initiate credential requests.
- **Issuance of Privacy-Preserving Credentials**: PSM issues p-ABCs based on attributes, enabling selective disclosure and privacy-preserving authentication.
- **Unlinkable Token Generation**: Tokens derived from p-ABCs allow entities to access services without linkability across sessions or services.
- **Dynamic Resource Protection**: Access to services is mediated by dynamic protection mechanisms (e.g., privacy-preserving proxies, ABAC policies) governed by the PSM.
- **Attribute-Based Access Control (ABAC)**: Fine-grained access decisions enforced through dynamically evaluated attribute-based policies.
- **Blockchain-Backed Trust**: Credential issuance, policy updates, and revocations are immutably recorded on blockchain, ensuring traceability and auditability.

6.2 Association Setup

This integration scenario highlights the capabilities of the initial release of the EMPYREAN platform to support the creation of Associations, enabling the establishment of collaborative virtual execution environments across heterogeneous edge and cloud platforms. This corresponds to operation flow OF2.1, which is executed by EMPYREAN administrators and authorized infrastructure providers with the appropriate permissions. In accordance with EMPYREAN's generic operation flow, these actions are preceded by the initialization of the platform by the administrator, including the deployment of core services such as the EMPYREAN Registry and Identity and Authorization Engine. Additionally, the initial stakeholders must be enrolled in the system, as described in Section 6.1.

Table 9 provides an overview of this operation flow, detailing the involved EMPYREAN components, relevant interfaces, coverage of platform requirements, and the enabling project technologies that support its execution.

Table 9: Overview of Association setup operation flow

EMPYREAN Components	EMPYREAN Registry: ● *API Gateway*: http://147.102.16.114:30150 ● *Service Catalogue:* http://registry-service-catalogue.empyrean:10090 ● *Association Metadata Store:* http://registry-association-metadata-service.empyrean:10086 EMPYREAN Aggregator 1: http://147.102.16.114:30800 EMPYREAN Aggregator 2: http://147.102.16.115:30800 *Dashboard*: http://147.102.22.140:8080
Type of APIs	REST
Requirements Coverage	F_GR.1, F_GR.2, F_GR.4, F_GR.5, F_ST.1, F_ST.2, F_SO.6, F_ASSOC.1, F_ASSOC.8, F_ASSOC.10
Enablers	EN_1, EN_9, EN_10, EN_11

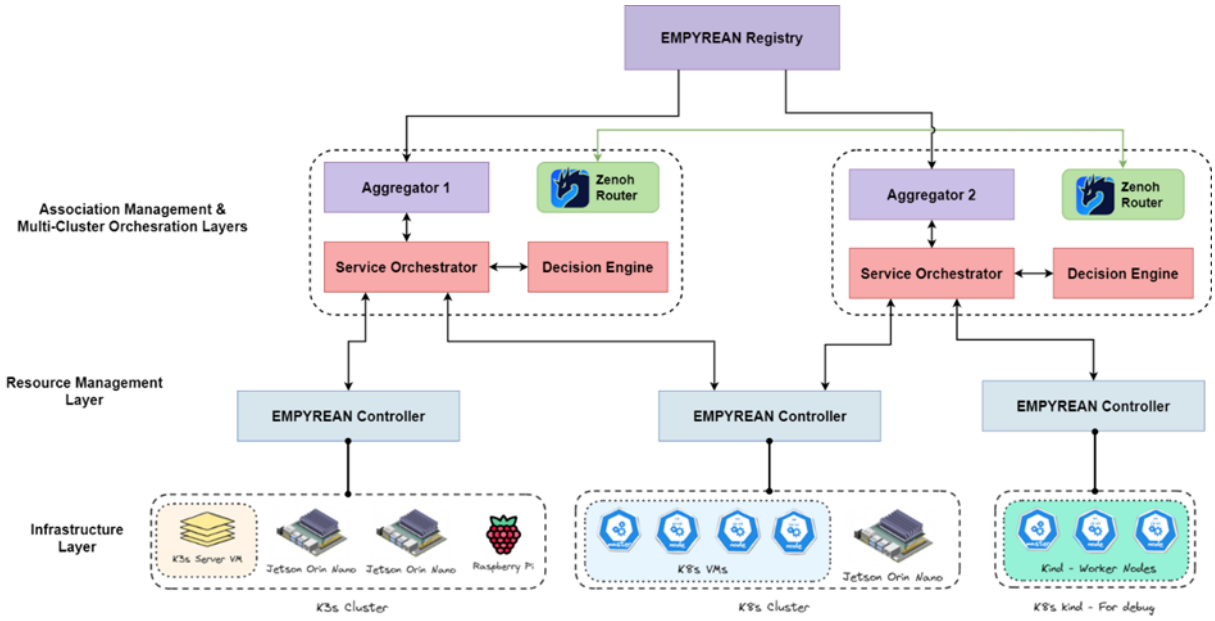


Figure 63: EMPYREAN components and testbed setup for the Association setup operation flow.

The integration scenario encompasses two Kubernetes (K8s) clusters and one K3s cluster, each featuring distinct characteristics, see Figure 63. The EMPYREAN Registry was deployed using EMPYREAN's CI/CD mechanisms on the ICCS K8s cluster. While the Registry API Gateway is publicly accessible, access is restricted through authentication mechanisms. The internal Registry services (Service Catalogue and Association Metadata Store) are configured for internal cluster access but remain accessible externally through the API Gateway. The setup also includes two EMPYREAN Aggregators, the first (UUID *"a2b66fa3-3f13-416b-8644-56328e0de4ba"*) is deployed on the ICCS K8s cluster, while the second (UUID *"d09ac308-4b4e-4623-9d75-a2633f229c7f"*) on the other K8s cluster.

In addition, a web-based application written in Python was developed to facilitate the demonstration of this integration scenario (Figure 64). Deployed within the ICCS premises, the application leverages the exposed REST APIs of the EMPYREAN components to interact with the platform. It also provides a visual representation of the available Associations and the onboarded devices.

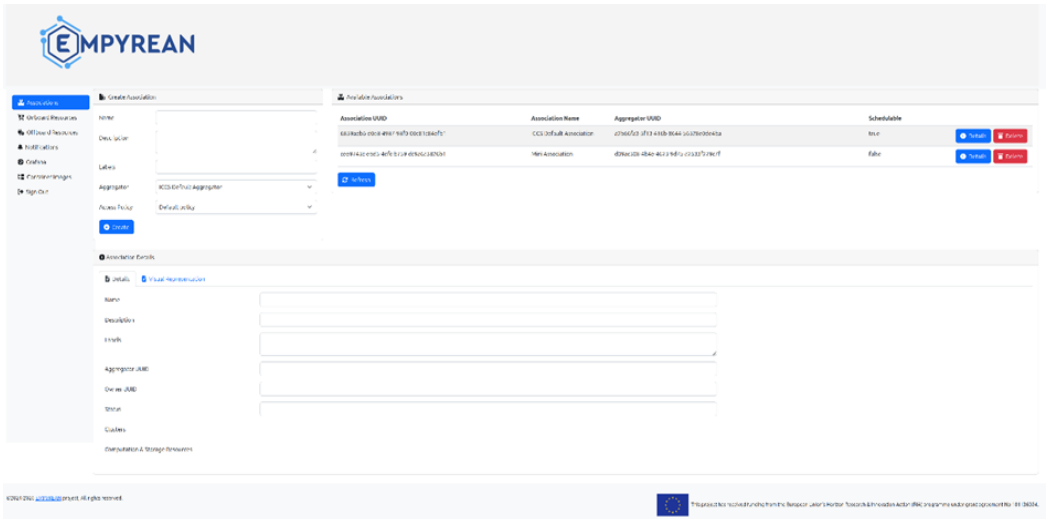


Figure 64: EMPYREAN web-based dashboard.

Using our web-based application, we created two Associations owned by the same user. The main difference is that, in the next phase (Section 6.3), different sets of resources will be onboarded to each of them. An administrator enrolled through the previous operation flow with Verifiable Credentials and JSON Web Token (JWT) access token logins to the dashboard.

When creating an Association, the user must select an Aggregator to manage it and link it to an existing access policy. In addition to these key parameters, which affect usage and resource management, optional configuration settings support integration with other platform services. These include the Association's name, an optional description, and user-defined labels used to identify Associations based on specific criteria.

Figure 65 presents the two Associations along with their corresponding configuration parameters as defined for the EMPYREAN control plane mechanisms.


```
{
  "name": "ICCS Association",
  "description": "The default Association based on resources from the ICCS testbed"
  "lables": ["platform-arch:amd64", "platform-arch:arm64"],
  "aggregator_uuid": "a2b66fa3-3f13-416b-8644-56328e0de4ba",
  "owner_uuid": "71c1642b-69c9-4e25-abcc-dda116e8becd",
  "policy_uuid": "5c0eed16-31a8-467d-83db-ef1010466755"
}
```

```
{
  "name": "Mini Association",
  "description": "The default Association based on resources from the ICCS testbed"
  "lables": ["platform-arch:arm64", "device_label:jetson"],
  "aggregator_uuid": "d09ac308-4b4e-4623-9d75-a2633f229c7f",
  "owner_uuid": "71c1642b-69c9-4e25-abcc-dda116e8becd",
  "policy_uuid": "5c0eed16-31a8-467d-83db-ef1010466755"
}
```


Figure 65: Association description parameters for initial integration scenarios.

The provisioning process is orchestrated by the EMPYREAN Registry. To initiate it, the dashboard invokes the Registry's POST method (*/api/v1/registry/associations*), which triggers the end-to-end Association creation procedure.

Upon receiving the request, the EMPYREAN Registry API Gateway performs an initial validation to ensure that all required information is provided. It then delegates the authorization process to the Privacy and Security Manager (PSM), specifically the Policy Decision Point (PDP) component. The PDP evaluates the request against predefined policies to verify that the user has the appropriate permissions to perform the operation. A log screenshot from the API Gateway illustrating these interactions is provided below. The red highlighted box shows the interactions between the dashboard and the API Gateway for retrieving available Associations, Aggregators, and access policies. The blue highlighted box depicts the involvement of the API Gateway during the creation of a new Association.

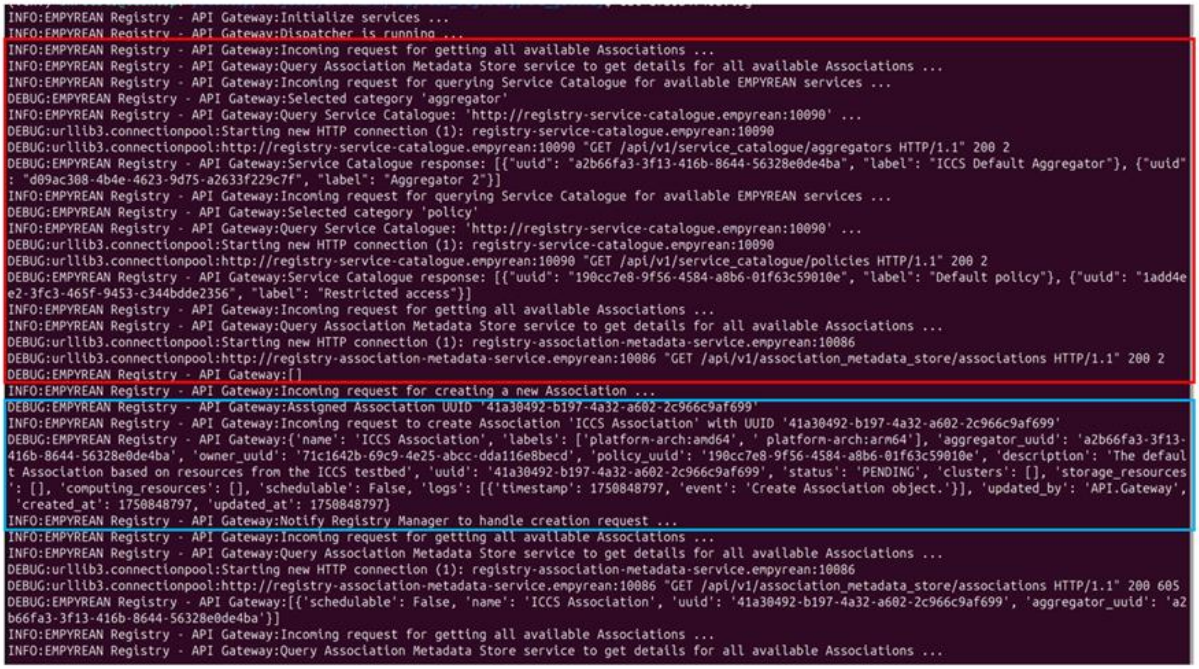


Figure 66: EMPYREAN Registry - API Gateway log messages during the definition of a new EMPYREAN Association from the dashboard.

Once authorization is successfully granted, control is handed over to the Registry Manager service, which coordinates all subsequent steps to set up the Association. It begins by creating an Association object in the internal Datastore, capturing the initial configuration parameters of the new Association. Next, the Registry Manager stores the Association's metadata, along with essential operational parameters, into the Association Metadata Store service by calling the relevant POST method *(/api/v1/association_metadata_store/associations*) exposed by its REST API. At this stage, the Metadata Manager component populates the graph database with the following nodes: (i) Association, (ii) Label(s), if specified, (iii) Aggregator, (iv) Owner, and (iv) Policy. Moreover, it establishes the following relationships: (i) Association to Aggregator, (ii) Association to Label(s), If specified, (iii) Association to Owner, and (iv) Association to Policy

Figure 67 provides a visual representation of the graph-based internal mapping of Associations within the EMPYREAN platform following the successful creation of the first Association

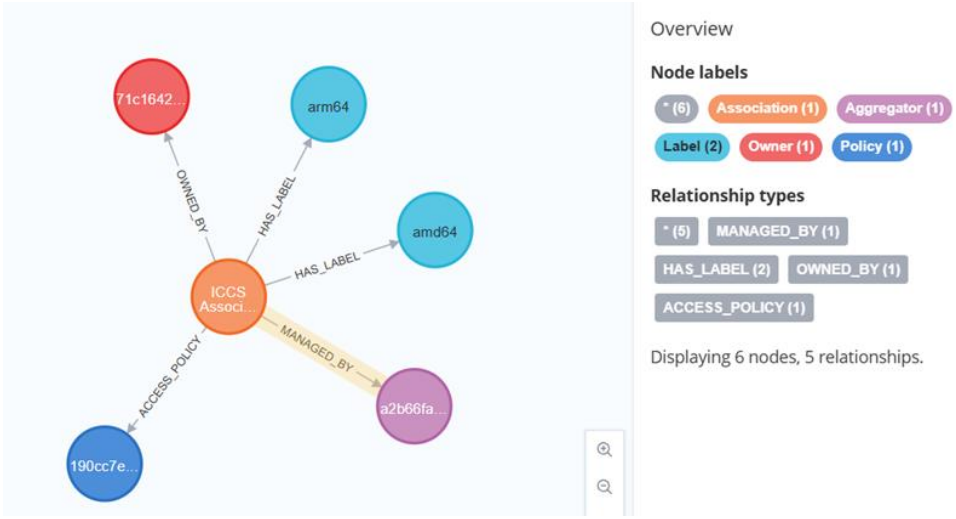


Figure 67: Graph-based representation of the first Association within the Association Metadata Store service.

The next step involves assigning the newly created Association to an EMPYREAN Aggregator for management. In the current release of the EMPYREAN platform, only existing Aggregators can be assigned. However, the final release will also support the automatic deployment of a new Aggregator, along with the corresponding Association assignment.

To perform the assignment, the Registry Manager invokes the corresponding POST method (*/api/v1/aggregator/associations*), which instructs the designated Aggregator to register the new Association in its State Management service. From that point onward, the Aggregator autonomously manages the Association, abstracting the complexities of interacting with its underlying platform mechanisms. Furthermore, the Aggregator informs the Telemetry Service and the related Analytics Engine about the new Association, enabling relevant data collection and monitoring processes to begin. Once this step is completed, the Association is successfully registered within the platform and marked as “CREATED”. However, its schedulable property remains False, as no resources have been onboarded yet.

Figure 68: A log screenshot from the Registry Manager capturing all these interactions is provided below.

Figure 69 illustrates the details for the two created Associations within the Association Metadata Store.

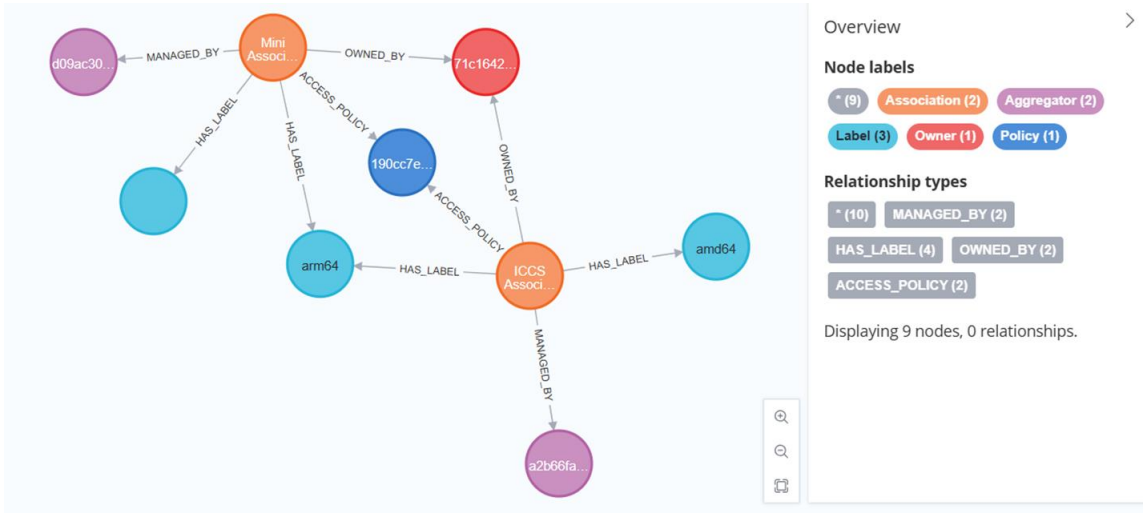


Figure 69: Availability of two created Associations within the dashboard and their graph-based representation.

6.3 Onboarding Computational and Storage Resources

Following the successful setup of Associations within the EMPYREAN platform, the next step involves the onboarding of computational and storage resources to make these Associations operational. This process enables the dynamic inclusion of diverse resource types, ranging from Kubernetes (K8s/K3s) worker nodes to IoT devices and edge storage systems, into the virtual execution environments defined by the Associations. The onboarding operation flows ensure that the EMPYREAN platform can securely and efficiently extend its control and service orchestration and deployment capabilities across heterogeneous infrastructure components.

This section details the implemented procedures within the initial release for onboarding:

- Worker nodes from Kubernetes and K3s clusters, which contribute containerized compute capacity to the Association.
- Edge storage resources, enabling localized data persistence and content availability within the Association scope.

Table 10: Overview of onboarding computational and storage resources operation flows.

EMPYREAN Components	EMPYREAN Registry API Gateway: http://147.102.16.114:30150 Aggregator 1: • API Gateway: http://147.102.16.114:30800 • Service Orchestrator: http://147.102.16.114:30100 Aggregator 2: • API Gateway: http://147.102.16.115:30800 • Service Orchestrator: http://147.102.16.115:30100 An EMPYREAN Controller at each cluster Edge Storage Service Dashboard: http://147.102.22.140:8080
Type of APIs	REST
Requirements Coverage	F_GR.1, F_GR.4, F_GR.6, F_ASSOC.1, F_ASSOC.2, F_ASSOC.7, F_ASSOC.8, F_ASSOC.9, F_ASSOC.7, F_DCM.1, F_DI.4, F_SO.4, F_SO.8, F_SO.14, F_ST.1, F_ST.2, F_DI.6
Enablers	En_1, EN_2, EN_5, EN_9, EN_10, EN_11, EN_16

6.3.1 Onboarding worker nodes

This integration scenario utilizes the testbed and the EMPYREAN platform services introduced in Section 6.2. In addition to those, the scenario also includes two Service Orchestrators, one per Aggregator, along with three EMPYREAN Controllers, each managing a distinct cluster (ICCS K8s “7628b895-3a91-4f0c-b0b7-033eab309891”, ICCS K3s “2b05cfdf-679f-45f1-95f8-a334ec87faaf”, and K8s-kind “5a075716-7d7d-4b40-9566-bc1a33ee70c2”).

The initial release of the EMPYREAN platform supports the static onboarding of worker nodes into Associations. This process must be initiated by either an EMPYREAN administrator or an infrastructure provider, both of whom must first be identified and authenticated as resource owners via the Security and Privacy Manager. To ensure the integrity of the onboarding process, the EMPYREAN Controller facilitates secure node registration. Each worker node (e.g., Kubernetes/K3s node) must authenticate before being admitted into an Association, ensuring identity verification and trustworthiness of resources.

The onboarding process can be triggered either through the EMPYREAN dashboard or via exposed REST APIs. The authorized user selects the target Association and designates which worker nodes to onboard. Figure 70 presents an overview of the selected nodes from the three available clusters and their mapping to the two Associations.

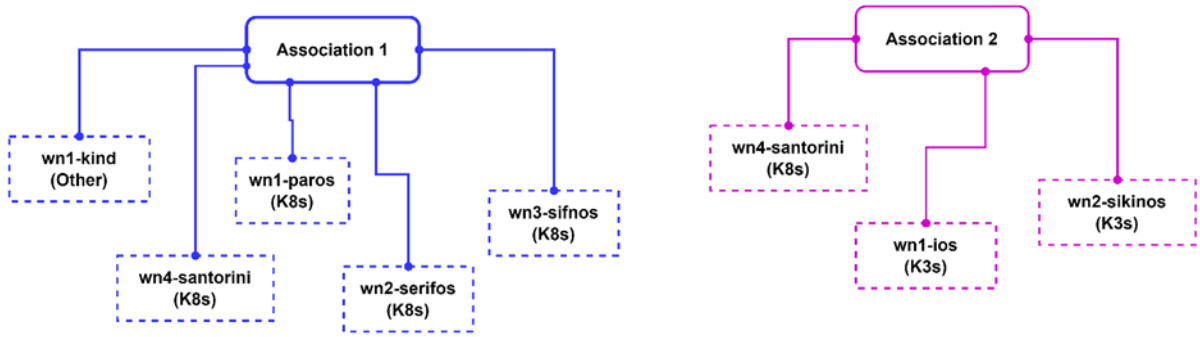


Figure 70: Mapping of selected worker nodes to Associations during the onboarding process.

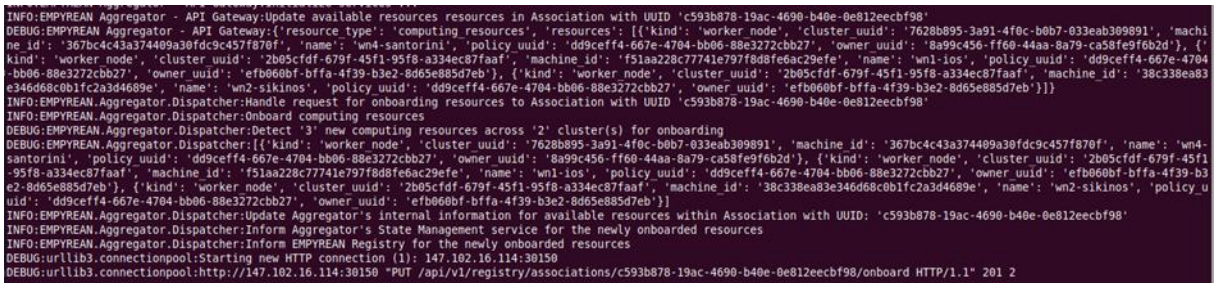
Using the dashboard (Figure 71), the user selects an appropriate EMPYREAN Controller based on access policies. The Controller then presents the available worker nodes eligible for onboarding. Once selection is complete, the onboarding is triggered, under the hood, this corresponds to a PUT request to the Aggregator API Gateway (*/api/v1/aggregator/ASSOCIATION_UUID/resources*).

Figure 71: Onboarding worker nodes through the EMPYREAN dashboard.

Next, we present the exact REST payload sent during the onboarding of resources for the first Association, along with a log excerpt from the Aggregator responsible for managing the second Association (c593b878-19ac-4690-b40e-0e812eecbf98), illustrating the internal handling of the onboarding process.


```
{"resource_type":"computing_resources","resources":[{"kind":"worker_node","name":"wn1-paros","cluster_uuid":"7628b895-3a91-4f0c-b0b7-033eab309891","machine_id":"02e257de949c4486b85ba436ec983663","policy_uuid":"dd9ceff4-667e-4704-bb06-88e3272cbb27","owner_uuid":"8a99c456-ff60-44aa-8a79-ca58fe9f6b2d"},{"kind":"worker_node","cluster_uuid":"7628b895-3a91-4f0c-b0b7-033eab309891","name":"wn2-serifos","machine_id":"dc156c5469db44c0be8121e8b94e31f6","policy_uuid":"dd9ceff4-667e-4704-bb06-88e3272cbb27","owner_uuid":"8a99c456-ff60-44aa-8a79-ca58fe9f6b2d"},{"kind":"worker_node","cluster_uuid":"7628b895-3a91-4f0c-b0b7-033eab309891","machine_id":"dc156c5469db44c0be8121e8b94e31f6","name":"wn3-sifnos","policy_uuid":"dd9ceff4-667e-4704-bb06-88e3272cbb27","owner_uuid":"8a99c456-ff60-44aa-8a79-ca58fe9f6b2d"},{"kind":"worker_node","cluster_uuid":"7628b895-3a91-4f0c-b0b7-033eab309891","machine_id":"367bc4c43a374409a30fdc9c457f870f","name":"wn4-santorini","policy_uuid":"dd9ceff4-667e-4704-bb06-88e3272cbb27","owner_uuid":"8a99c456-ff60-44aa-8a79-
```

ca58fe9f6b2d"},{"kind":"worker_node","cluster_uuid":"5a075716-7d7d-4b40-9566-bc1a33ee70c2","machine_id":"4e06de61977c4c8aaf9c1e9bf239aaab","name":"wn1-kind","policy_uuid":"dd9ceff4-667e-4704-bb06-88e3272cbb27","owner_uuid":"8a99c456-ff60-44aa-8a79-ca58fe9f6b2d"}]}



Once the onboarding process is complete, the worker nodes are fully integrated into the Association and become available for workload assignments, contributing to the Association's overall computational and operational capacity. Finally, Figure 72 visualizes the updated state of both Associations as recorded in the Association Metadata Store service within the EMPYREAN Registry.

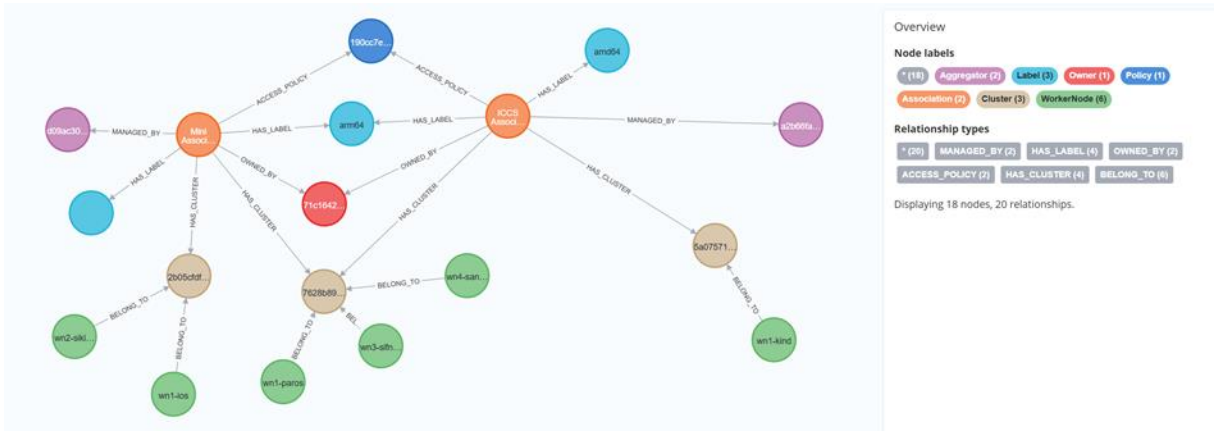


Figure 72: The two Associations along with the onboarding worker nodes within Association Metadata Store.

6.3.2 Onboarding storage resources

The Edge Storage Service (ESS) is an integrated part of the initial EMPYREAN platform release. It manages storage entities such as S3-compatible buckets and objects, as well as some of the storage-related resources and infrastructure. Since Associations, including their associated resources, are handled by the Registry, the ESS periodically polls this component for any changes through the API described in Section 5.2.1. Figure 73 shows two Associations that have been retrieved from the EMPYREAN Registry and stored in the ESS's database for authorizing user requests.

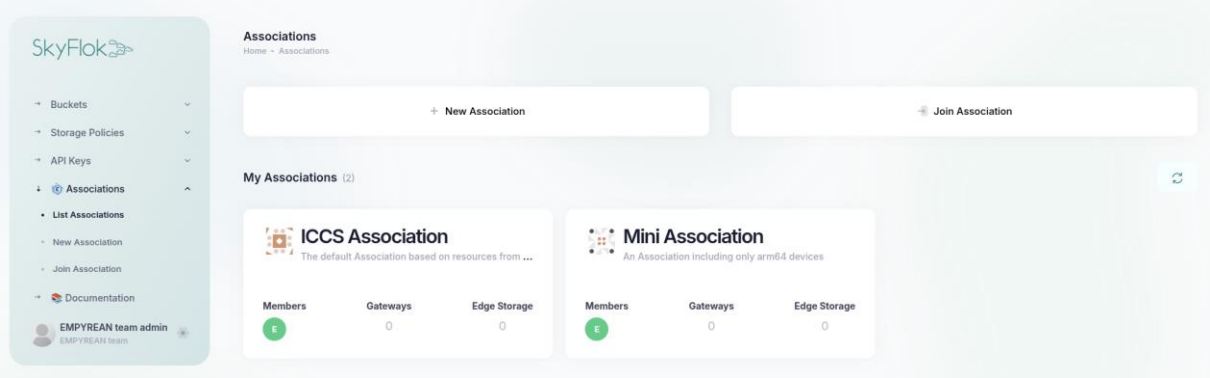
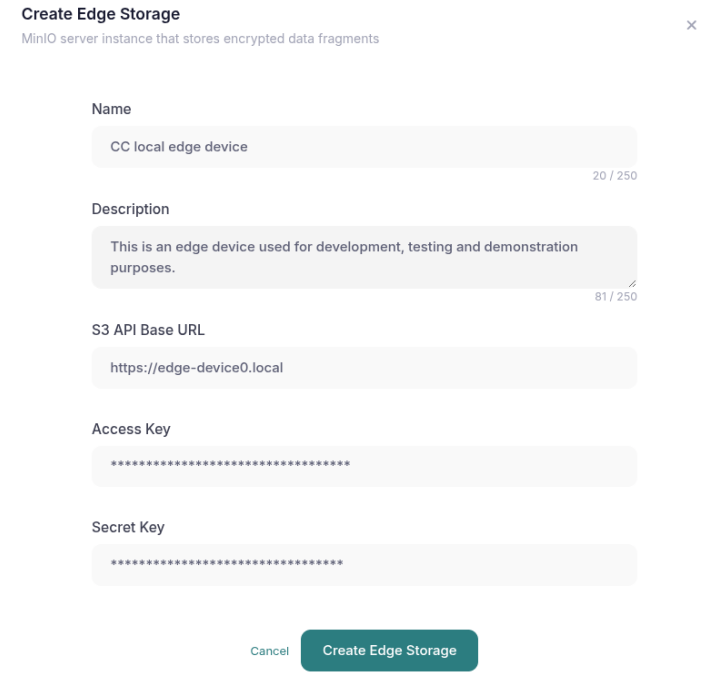


Figure 73: Edge Support Service Developer Dashboard: two associations retrieved from the EMPYREAN Registry.

Once the Association Setup workflow is complete and the ESS has retrieved the list of Associations, storage resources can be onboarded. In particular, the Edge Storage devices can be registered using a form on the ESS Developer Dashboard, as shown in Figure 74. The Association's owner/administrator must specify the location of the storage resource and what credentials can be used to access it. These credentials are persisted by the ESS, but are not shared with any of the Association's members. Instead, the access is performed using pre-signed URLs [10].



.

Figure 74: Edge Support Service Developer Dashboard: form for registering a new Edge Storage device.

The ESS notifies the EMPYREAN Registry that a new edge resource has been onboarded for the association in question using the API described in Section 5.2.1.

In the following, we describe a typical storage workflow from the perspective of an EMPYREAN user. We have defined the workflow to demonstrate the level of integration at the time of the platform's initial release as well as to provide an initial assessment of how well different storage-related requirements are met.

To make the demonstration robust, we have set up an Association with edge resources running on a single laptop. This allows us to quickly deploy both an Edge Storage Gateway and several Edge Storage devices.

1. As a first step, we list the Association's storage resources as shown in Figure 75.

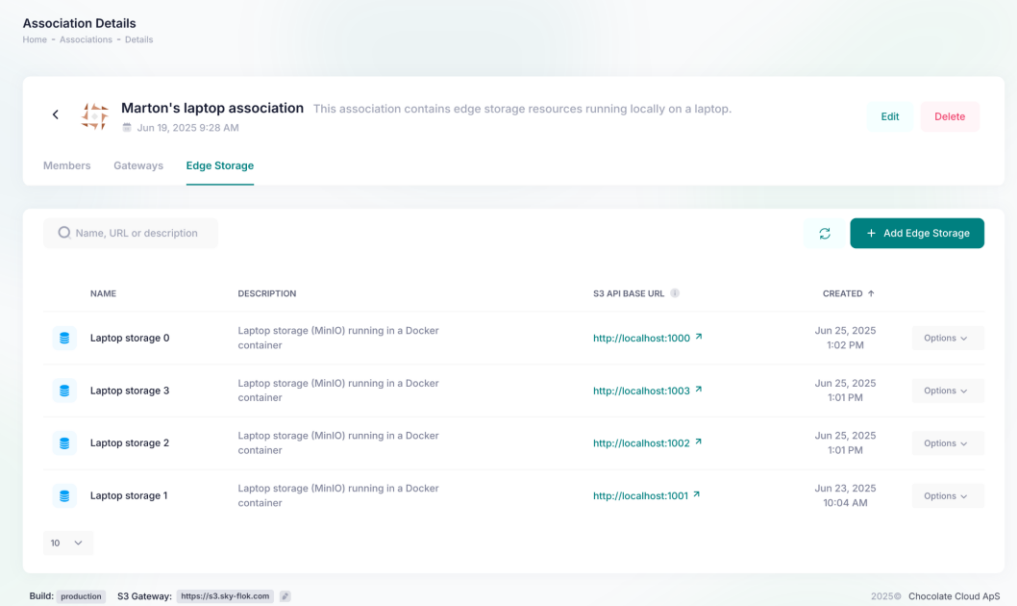


Figure 75: Step 1 - list of Edge Storage devices running in the Association.

2. We can use these to create an edge-only storage policy. Figure 76 shows the result, an erasure coded policy with 33% redundancy that distributes data to all four Edge Storage devices.

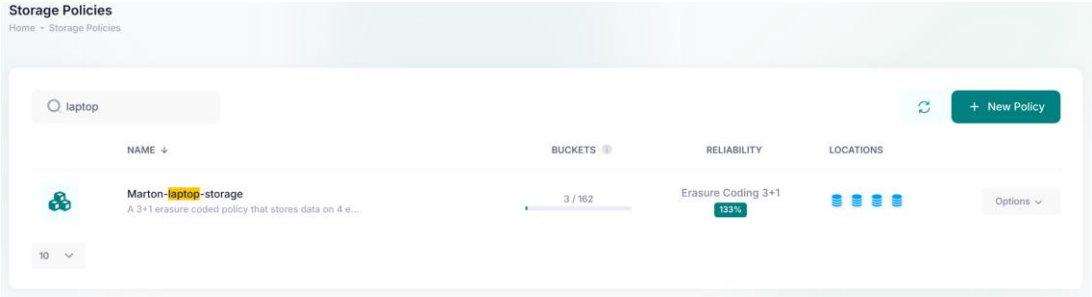


Figure 76: Step 2 - summary of edge-only storage policy.

3. At this point, we have registered all resources and created an initial storage policy. To use the ESS, we must deploy an Edge Storage Gateway and the four Edge Storage devices within our Association. For this demonstration, we accomplish this using Docker Swarm. Figure 77 shows the result. The Gateway must be associated with the Association and has been set up using the Developer Dashboard (not shown for brevity).

```
marton@Blade:~/repo/SKYFLOK/skyflok-s3-gateway$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS          PORTS
                                                                                      NAMES
b6c44340115b   minio-eval3    "/usr/bin/docker-ent…"   26 seconds ago   Up 25 seconds   0.0.0.0:1003->1003/tc
p, [::]:1003->1003/tcp, 0.0.0.0:9003->9003/tcp, [::]:9003->9003/tcp, 9000/tcp   practical_swartz
2742f8f14f82   minio-eval2    "/usr/bin/docker-ent…"   26 seconds ago   Up 25 seconds   0.0.0.0:1002->1002/tc
p, [::]:1002->1002/tcp, 0.0.0.0:9002->9002/tcp, [::]:9002->9002/tcp, 9000/tcp   dreamy_satoshi
b754b839433d   minio-eval1    "/usr/bin/docker-ent…"   26 seconds ago   Up 25 seconds   0.0.0.0:1001->1001/tc
p, [::]:1001->1001/tcp, 0.0.0.0:9001->9001/tcp, [::]:9001->9001/tcp, 9000/tcp   blissful_chatelet
afb220eef8f4   minio-eval0    "/usr/bin/docker-ent…"   26 seconds ago   Up 26 seconds   0.0.0.0:1000->1000/tc
p, [::]:1000->1000/tcp, 0.0.0.0:9000->9000/tcp, [::]:9000->9000/tcp             brave_banzai
45347cf555ae   a2f0faaac4eb   "uvicorn main:app --…"   12 days ago      Up 6 seconds
                                                                                      skyflok-s3-gateway
```

Figure 77: Step 3 - four Edge Storage devices and the Edge Storage Gateway running locally on a laptop.

4. To access the service and sign our application's requests, we need a set of credentials. In S3, this consists of a pair of items (Access Key ID and Secret Access Key) that we collectively refer to as an S3 API key. This can be done using the Developer Dashboard (not shown for brevity). We can create a non-restricted S3 API key that can access any of the team's buckets, or we can permit read-only or full access on a per-bucket basis.

5. Finally, the application must be configured to use the Gateway's URL and the S3 API keys (Figure 78 - left).

6. The application can make S3 API requests to the ESS, through the Gateway. A simple script (Figure 78 - left) and its resulting output (Figure 78 - right) have been included.

```python
import os
import boto3
from botocore.client import Config


# Set up variables
GATEWAY_URL = "http://localhost:8080"
ACCESS_KEY = "SKYVKQNNIBA6BDXKU8JMMDIFXRCUQHUK"
SECRET_KEY = "y8TRmHlyoC6dO6X57BtESEPoHnZnwPQAAmLC0tt5dLlu60dr4rfd7Qxz8l8XBzwl"

BUCKET_NAME = "my-third-demo"
OBJECT_KEY = "test-object"

config: Config = Config(
            signature_version='s3v4',
            retries = {
                'max_attempts': 1,
                'mode': 'standard'
            })

# Boto3 S3 client
s3: _ | Any = boto3.client(
    's3',
    endpoint_url="{gateway_url}".format(gateway_url=GATEWAY_URL),
    config=config,
    aws_access_key_id=ACCESS_KEY,
    aws_secret_access_key=SECRET_KEY,
    verify=False
)

print("\n*** Listing buckets ***")

# List buckets
resp: Any = s3.list_buckets()
for bucket in resp['Buckets']:
    print(f"Bucket name: {bucket['Name']}")

# Upload an object with 100bytes of random data
file_data: bytes = os.urandom(100)
resp: Any = s3.put_object(Body=file_data, Bucket=BUCKET_NAME, Key=OBJECT_KEY,
                          Metadata={'string-a': 'string'},)

# Check it has been uploaded
resp: Any = s3.list_objects_v2(Bucket=BUCKET_NAME, MaxKeys = 200)

print("\n*** Listing objects in bucket ***")

if resp["KeyCount"]>0:
    for obj in resp['Contents']:
        print(f"Key: {obj['Key']}")
        print(f"Size: {obj['Size']}")
        print(f"Last modified: {obj['LastModified']}")
```

```
marton@Blade:~/repo/EMPYREAN$ python3 gent_demo.py

*** Listing buckets ***
Bucket name: my-third-demo

*** Listing objects in bucket ***
Key: test-object
Size: 100
Last modified: 2025-07-14 14:33:47+00:00
marton@Blade:~/repo/EMPYREAN$
```

Figure 78: Steps 5 and 6 - Simple Python script (left) that uses Amazon's Boto3 library to access the ESS and the results of running the script (right).

6.4 Inter-Association Application Deployment

This operation flow highlights EMPYREAN's capability to deploy cloud-native applications across federated Associations. Application operators manage deployments using developer-provided descriptions that define resource and configuration requirements. Deployment follows a structured, multi-phase orchestration process. In Phase 1 (OF4.1.1), decentralized orchestration selects the most suitable Associations based on deployment goals. Phase 2 (OF4.1.2) performs cognitive orchestration within each Association, refining deployment plans based on local policies and infrastructure conditions. Finally, in the third phase (OF4.1.3), the deployment is executed at the infrastructure level by local orchestrators. This section focuses on the first two orchestration phases. Summarised in Table 11

Table 11: Overview of operation flow for inter-association application deployment.

EMPYREAN Components	EMPYREAN Registry API Gateway: http://147.102.16.114:30150 Aggregator 1 API Gateway: http://147.102.16.114:30800 Aggregator 2 API Gateway: http://147.102.16.115:30800 Service Orchestrator 1: http://147.102.16.114:30100 Service Orchestrator 2: http://147.102.16.115:30100 Decision Engine 1: http://147.102.16.114:30020 Decision Engine 2: http://147.102.16.115:30020 Telemetry Service: http://147.102.16.114:30070
Type of APIs	REST
Requirements Coverage	F_GR.1, F_GR.2, F_GR.3, F_GR.4, F_GR.5, F_GR.6, F_ASSOC.1, F_ASSOC.5, F_ASSOC.8, F_ASSOC.9, F_ASSOC.10, F_ST.2, F_DI.1, F_DI.2, F_DI.3, F_SO.2, F_SO.3, F_SO.4, F_SO.5, F_SO.6, F_SO.9, F_SO.14
Enablers	EN_2, EN_4, EN_9, EN_10, EN_11, EN_14, EN_17

This scenario builds on testbed and the EMPYREAN platform services introduced in Section 6.2. A toy application, composed of five microservices (Figure 79), is used to demonstrate end-to-end integration. The application collects data, performs quality inference, stores results, and includes a retraining trigger based on inference outcomes. A simple web UI allows users to view outputs. Each microservice serves a specific role in processing apple quality data, using datasets and models were sourced from Kaggle's Apple Quality Dataset [11]. Deployment is managed via K8s\K3s YAML descriptions, defining key components Deployment, ConfigMap, Service, and PersistentVolume. Figure 79 maps these objects to each microservice

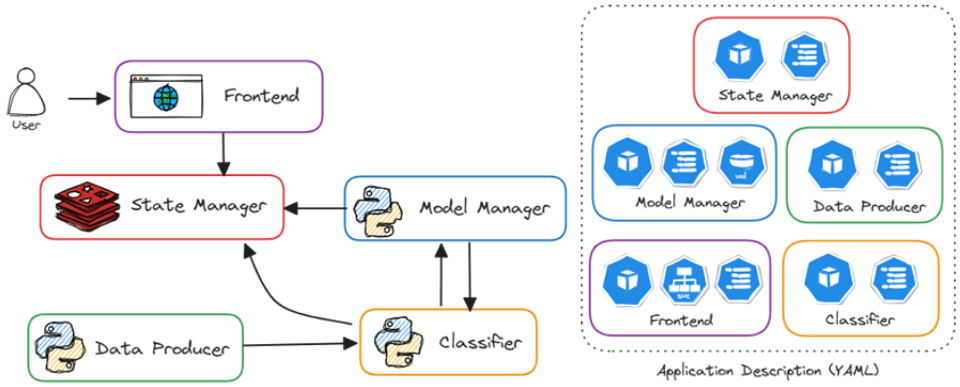


Figure 79: Toy cloud-native application for inter-Association deployment demonstrator.

OF4.1.1: Assignment of application's microservices to EMPYREAN Associations

The application operator initiates this operation flow by sending the appropriate POST method in their designated Aggregator (*Step 1*), including also the application description along with any relevant deployment objectives. In the validation scenario, Aggregator 1 (UUID: "*a2b66fa3-3f13-416b-8644-56328e0de4ba*") is used. Upon receiving the request, the Aggregator API Gateway validates the input parameters and proceeds to create the corresponding *Application* and *EMPYREAN Deployment* API objects in its Datastore (*Steps 2-3*). The Aggregator then responds with the unique EMPYREAN deployment identifier "23b4065c-f4cb-4bb4-b06d-045f831cdad9" for reference.

```
INFO:EMPYREAN.Aggregator.APIGateway:Initialize services ...
INFO:EMPYREAN.Aggregator.APIGateway:Incoming request for new Associations Deployment ...
DEBUG:EMPYREAN.Aggregator.APIGateway:Assigned Associations Deployment UUID '23b4065c-f4cb-4bb4-b06d-045f831cdad9'
INFO:EMPYREAN.Aggregator.APIGateway:Handle Associations Deployment request with UUID '23b4065c-f4cb-4bb4-b06d-045f831cdad9'
                                                                                                                    Step 1
INFO:EMPYREAN.Aggregator.APIGateway:Create Application object ...
INFO:EMPYREAN.Aggregator.APIGateway:Create EMPYREAN Deployment object ...                                           Steps 2 &3
DEBUG:EMPYREAN.Aggregator.APIGateway:{'uuid': '23b4065c-f4cb-4bb4-b06d-045f831cdad9', 'kind': 'EMPYREANDeployment', 'name': 'debug-of4.1', 'originated_association_uuid': '6839aeb6
-d0a8-4987-98f0-00c81c84efb1', 'user_uuid': '', 'application_uuid': '7c037ee7-fc3e-45c7-ad46-3cc7bc847e45', 'multi_agent_deployment_uuid': None, 'associations': [], 'association_d
eployments': [], 'association_deployments_status': [], 'logs': [{'timestamp': 1753180575, 'event': 'Deployment request received.'}], 'status': <AssociationsDeployment.SUBMITTED: 1
>, 'updated_by': 'Aggregator.API', 'created_at': 1753180575, 'updated_at': 1753180575}
INFO:EMPYREAN.Aggregator.APIGateway:Store Application object '7c037ee7-fc3e-45c7-ad46-3cc7bc847e45' for Associations Deployment '23b4065c-f4cb-4bb4-b06d-045f831cdad9'
INFO:EMPYREAN.Aggregator.APIGateway:Store EMPYREAN Deployment object '5c9efea8-a7f9-4676-8ad7-30fe6a6bc0ea' for Associations Deployment '23b4065c-f4cb-4bb4-b06d-045f831cdad9'
```

The Aggregator Dispatcher service, which monitors the deployment-related topic (i.e, */empyrean/aggregator/a2b66fa3-3f13-416b-8644-56328e0de4ba/empyrean_deployments/deployment*) detects the update (*Step 4*). After interpreting the deployment request, it forwards it to its corresponding Service Orchestrator for further processing (*Step 5*).

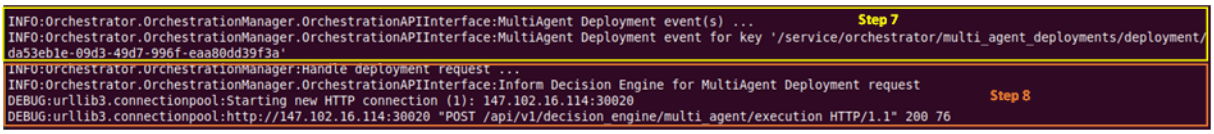
```
INFO:EMPYREAN.Aggregator.Dispatcher:Event key: '/empyrean/aggregator/a2b66fa3-3f13-416b-8644-56328e0de4ba/empyrean_deployments/deployment/5c9efea8-a7f9-4676-8ad7-30fe6a6bc0ea'
INFO:EMPYREAN.Aggregator.Dispatcher:Retrieve application details for requested deployment
INFO:EMPYREAN.Aggregator.Dispatcher:Inform corresponding Service Orchestrator for the Associations Deployment request with UUID '5c9efea8-a7f9-4676-8ad7-30fe6a6bc0ea'
DEBUG:urllib3.connectionpool:Starting new HTTP connection (1): 147.102.16.114:30100
DEBUG:urllib3.connectionpool:http://147.102.16.114:30100 "POST /api/v1/service_orchestrator/associations/deployments HTTP/1.1" 201 2
INFO:EMPYREAN.Aggregator.Dispatcher:Service Orchestrator associations_deployment_uuid:da53eb1e-09d3-49d7-996f-eaa80dd39f3a                    Step 5
INFO:EMPYREAN.Aggregator.Dispatcher:Update internal EMPYREAN Deployment API object '5c9efea8-a7f9-4676-8ad7-30fe6a6bc0ea'
```

The Orchestration API Server receives the incoming request and creates the corresponding *MultiAgent Deployment* API object in its local Datastore (*Step 6*).

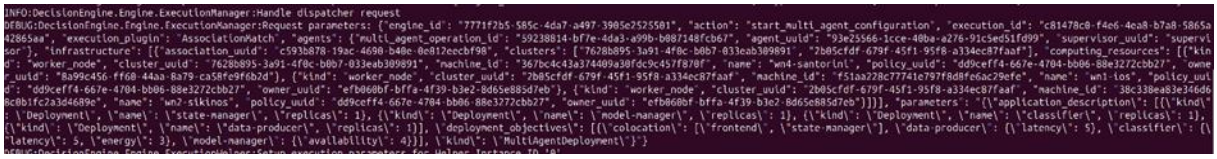
```
INFO:Orchestrator.OrchestratorAPI:Incoming request for new multi-agent deployment ...
DEBUG:Orchestrator.OrchestratorAPI:Assigned deployment uuid 'da53eb1e-09d3-49d7-996f-eaa80dd39f3a'
DEBUG:Service.Orchestrator.OrchestratorAPI:Create MultiAgent Deployment API object for deployment 'da53eb1e-09d3-49d7-996f-eaa80dd39f3a'
```

This action triggers the Orchestration Manager, which watches the topic */empyrean/orchestrator/multi_agent_deployments/deployment* for related events (*Step 7*). Upon activation, the Orchestration Manager, through its internal controllers, begins handling

the orchestration process. As a next step, the Orchestration Manager's Scheduler Controller invokes the Decision Engine to initiate a collaborative, multi-agent decision-making process across participating Aggregators (*Step 8*). This process aims to distribute the application's microservices across suitable Associations.



The orchestration request submitted to the Decision Engines includes both the deployment objectives and the application graph, which is parsed from the initial application description. Each Decision Engine independently queries the Telemetry Service to obtain real-time metrics for resources available within its respective Associations (*Step 9*). These outcomes are then forwarded to the designated coordinator Decision Engine to support collective decision-making. The following log excerpt, captured from Decision Engine 1, which manages Association "c593b878-19ac-4690-b40e-0e812eecbf98", demonstrates that the engine considers only the four worker nodes onboarded from the two clusters assigned to this Association.



Once consensus is reached, the coordinator Decision Engine notifies its local Service Orchestrator of the agreed resource allocation plan. The orchestrator then updates the originating Aggregator accordingly (Step 10).

OF4.1.2: Hierarchical and cognitive orchestration at the Association level

Once the application's microservices have been logically assigned to the appropriate Associations, the orchestration process advances to its second phase. This phase runs in parallel across all selected Associations and focuses on refining and localizing the deployment strategy according to the infrastructure and policies of each Association.

The initiating Aggregator triggers this phase by delegating the deployment responsibilities to the Aggregators managing the selected Associations. Within each participating Aggregator, the Aggregator API Gateway processes the relevant portion of the deployment plan and creates a corresponding Deployment API object in its local Datastore. This object contains only the microservices allocated to that Association, along with their associated configuration descriptors (e.g., ConfigMap, Secrets, etc.) extracted from the original application definition.

In the demonstration scenario, three microservices are assigned to the first Association, which is managed by the initiating Aggregator, while the remaining ones are delegated to the second Association. The following log illustrates how the Aggregator responsible for the second Association coordinates and progresses the orchestration process.

By the end of this phase, each Aggregator possesses a fully scoped deployment blueprint tailored to its respective Association, ready to be passed to the local deployment engine for execution in phase OF4.1.3.

7 Conclusions

Deliverable 5.2 provides a comprehensive overview of the first implementation cycle of the EMPYREAN platform, highlighting the development progress, integration activities, and early validation of core components. During this initial phase (M1–M18), the consortium successfully implemented core components, defined key interfaces, and delivered partial yet functional components across all architectural layers, including cloud-native development and deployment, telemetry, orchestration, AI-enabled services, and security.

Moreover, the deliverable presents the project's CI/CD processes along with the related development and integration guidelines. This structured approach aims to maximize the level of automation, thereby minimizing effort and risks during the integration of diverse modules and services developed by different partners.

The initial release demonstrates the feasibility of EMPYREAN's modular and federated approach to building an intelligent, self-managing IoT-Edge-Cloud continuum. Integration efforts have begun to align individual components to support interoperability within EMPYREAN Associations, and early validations have confirmed the viability of the core concepts and operation flows.

The results documented in this deliverable will guide the upcoming development and integration cycle, leading to the second release at M30. Feedback from internal testing and use case development will be used to further improve the platform's robustness, scalability, and AI-driven autonomy. The full integrated release, including all platform components and new features across all architecture layers, is planned for M30, following the conclusion of the second implementation iteration (M16-M26), which will be reported in deliverables D3.3 and D4.3 at M26.

For the final release of the EMPYREAN platform, delivered at the end of the project, the consortium will incorporate feedback from the final evaluation of the EMPYREAN platform through the demonstration of project use cases. This version will be fully integrated and documented in deliverables D5.4 *"Final release of EMPYREAN integrated platform"* (M36) and D6.2 "Demonstrators' deployment and EMPYREAN evaluation" (M36).

8 References

[1] EMPYREAN System requirements and specifications D2.1

[2] EMPYREAN System General Architecture Design D2.2

[3] Proxmox Platform. https://www.proxmox.com/en/

[4] EMPYREAN Code Source repository in GitHub. https://github.com/empyrean-eu

[5] Eclipse Zenoh. Official GitHub repository. (available online, visited July 2025) https://github.com/eclipse-zenoh/zenoh

[6] Zenoh-Flow Tutorial (available online, visited July 2025) https://www.youtube.com/watch?v=wGEM6-ByAL8

[7] EMPYREAN deliverable D4.2 Intelligent Resource Management, Cyber Threat Intelligence and EMPYREAN Aggregator.

[8] AWS-S3 API reference (available online, visited in July 2025):

https://docs.aws.amazon.com/AmazonS3/latest/API/API_Operations_Amazon_Simple_Storage_Service.html

[9] MinIO's Prometheus-compatible telemetry API (available online, visited in July 2025) https://min.io/docs/minio/linux/operations/monitoring/collect-minio-metrics-using-prometheus.html

[10] AWS - Amazon S3 pre-signed URLs (Available Online, visited on July 2025): https://docs.aws.amazon.com/AmazonS3/latest/userguide/ShareObjectPreSignedURL.html)

[11]Kaggle's Apple Quality Dataset: https://www.kaggle.com/datasets/nelgiriyewithana/apple-quality/data