A distributed data-mining software platform for extreme data across the compute continuum

D3.3 Data-Driven Orchestration and Monitoring (Final Release)

Version 1.0

Documentation Information

Contract Number	**101093110**
Project Website	www.extract-project.eu
Contractual Deadline	M30, 30th June 2025
Dissemination Level	Public
Nature	Report
Author	Barcelona Supercomputing Center (BSC)
Contributors	IBM, IKL, SIX
Reviewer	OBS
Keywords	Data, orchestration, monitoring, scheduling, workflow

Change Log

Version	Description Change
V0	Initial draft of the table of contents and collaboration assignments
V0.1	Initial BSC contributions
V0.2	Contributions from IKL
V0.3	Review (OBS)
V0.4	Apply comments from review
V0.9	Final revision (BSC)
V1.0	Ready to submit

The EXTRACT Project has received funding from the European Union's Horizon Europe programme under grant agreement number 101093110.

Table of Contents

1. Introduction

This document presents the final release of the Data-driven Orchestration and Monitoring system developed within Work Package 3 (WP3) of EXTRACT. Building upon the first release documented in Deliverable D3.2, this final version consolidates and enhances our previous developments, achieving a robust and fully integrated system that addresses the project's objectives related to optimized orchestration and monitoring across the compute continuum.

The main purpose of this deliverable is twofold: first, to clearly summarize the complete, finalized architecture and implementation details of our distributed monitoring infrastructure; and second, to explain the finalized orchestration approach that uses real-time collected metrics to efficiently schedule and deploy data-driven workflows.

Compared to D3.2, this document focuses on the following improvements and differences:

- **Enhanced Monitoring Architecture**:
  The monitoring system described in D3.2 has been further improved, now incorporating additional metrics, advanced data-processing capabilities, new exporters, and refined visualization techniques, ensuring better insights and increased reliability.
- **Optimized Workflow Orchestration**:
  The orchestration component has evolved significantly, integrating real-time monitoring data into scheduling algorithms. This integration enables more adaptive and efficient orchestration decisions compared to the heuristic-based approaches previously used.
- **Initial Validation and Analysis**:
  This deliverable also includes preliminary performance evaluation and analysis results, demonstrating the real-world efficiency and scalability of our solution in various scenarios. Most of the validation tests will be performed at the latest stage of the project, from M31 to M36.

1.1. Structure

This document is organized in 5 main sections:

- Section 1 – Introduction: Provides an overview of the scope of the deliverable and its relationship with other WPs and previous documents. It sets the context for the developments and evaluations reported.
- Section 2 – Final Distributed Monitoring Architecture: This section presents the finalized version of the EXTRACT monitoring architecture. It describes the federated components involved (storage, collectors, transport, visualization), the metrics implemented for different types of hardware (including Jetson-specific support), and how the architecture satisfies the monitoring requirements defined earlier in the project.

- Section 3 – Data-driven Workflow Deployment and Scheduling: Details the orchestration architecture with a focus on COMPSs as the main scheduling framework. It highlights the technological improvements made, such as horizontal and vertical scalability, integration with Kubernetes, and enhanced metric-driven scheduling. The section also introduces the finalized design for multi-cluster execution and its coupling with the monitoring stack.
- Section 4 – Evaluation and Performance Analysis: This section covers the ongoing evaluation of multicluster communication technologies. It explains the motivation behind the benchmarking activities, describes the experimental infrastructure setup across partner sites (OVH, Ikerlan, BSC), and outlines the methodology for testing. Preliminary steps are covered, while most testing is deferred to the final project stage (M30–M36).
- Section 5 – Conclusion: Summarizes the progress made in finalizing both the orchestration and monitoring architectures, and outlines the remaining work for the final months of the project.

1.2. Relationship with other WPs and Deliverables

Deliverable	Tasks	Relation
D1.3	T1.3	This deliverable integrates orchestration and monitoring enhancements supporting both TASKA and PER use-cases, specifically in data-driven workflow deployments, directly contributing to the workflow implementation documented in D1.3.
D2.3	T2.3	D3.3 utilizes the finalized data-mining framework described in D2.3 and provides orchestration and monitoring capabilities that ensure effective deployment, execution, and monitoring of workflows defined using the EXTRACT data-mining framework. In that deliverable, we also describe how COMPSs integrates Lithops jobs together.
D3.2	T3.2, T3.3	D3.3 is the final version and enhancement of D3.2, fully developing and integrating the orchestration mechanisms (T3.2) and the distributed monitoring infrastructure (T3.3) initially presented in D3.2.
D4.3	T4.2, T4.3	This deliverable leverages the final version of the compute continuum abstraction layer (D4.3), integrating the interoperability mechanisms (T4.2) and cybersecurity functionalities (T4.3) to provide robust and secure orchestration and monitoring across diverse computing environments.

Table 1. Relationship with other WPs

2. Final Distributed Monitoring Architecture

The distributed monitoring architecture was initially proposed in deliverable D3.1 and further extended in deliverable D3.2. The current section provides the evolution that the monitoring architecture has undergone for the current state. The different technologies used to implement each proposed components are defined and the different metrics that are being monitored are listed. Additionally, the current fulfillment of the requirements presented in deliverable D3.1 are specified which allows identifying the future steps towards the development of the monitoring architecture.

2.1. Architecture Overview

The monitoring architecture proposed in deliverable D3.1 and updated in deliverable D3.2 has been extended to support multicluster architectures as seen in Figure 1. Similar to how it was presented previously, this architecture will be deployed in Kubernetes using Ansible. However, in order to support multiple Kubernetes clusters needed for the use cases of this project, the architecture has been extended following a federated approach.

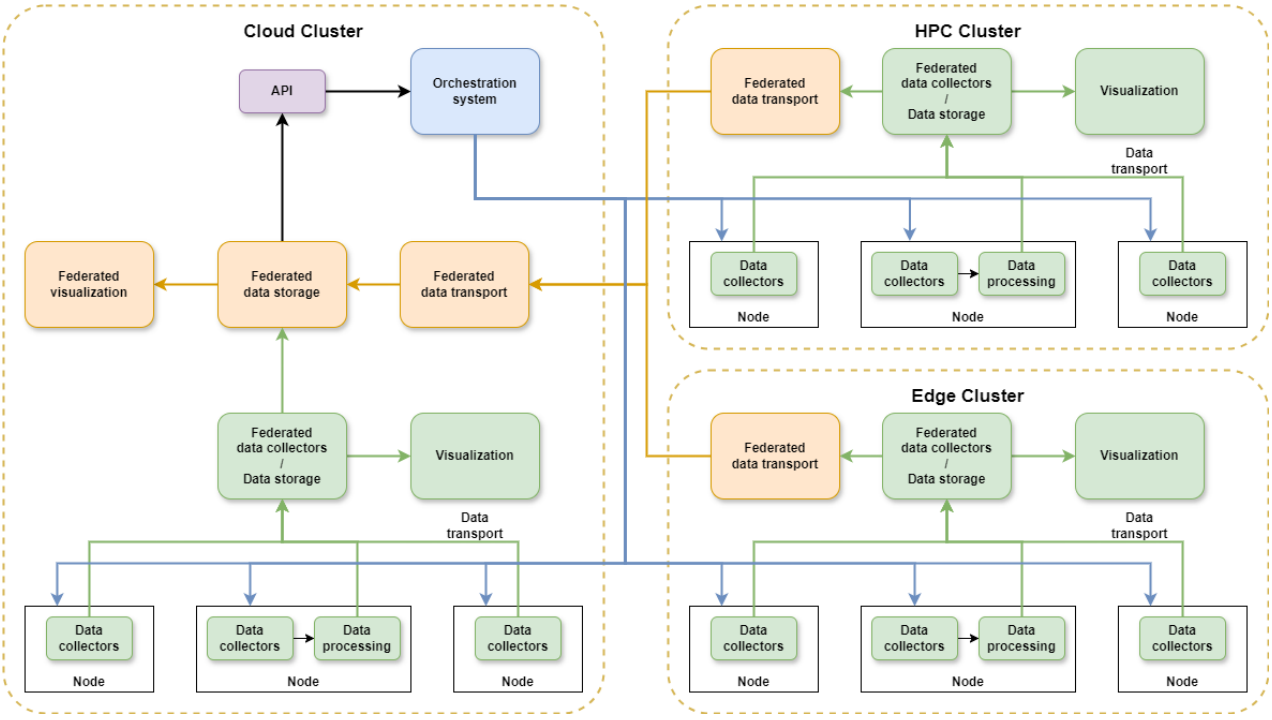

Figure 1 - Federated Monitoring Architecture

Each of the infrastructure clusters has its own Data Storage, Collectors, Transport, Processing Procedures and Visualization components. Additionally, one of the infrastructure clusters has Federated Data Storage, Transport and Visualization components, where each Data Storage from the clusters will act as Data Collectors for the federated system.

2.2. Federated Monitoring Components

The monitoring components for each of the clusters were described in deliverable D3.2. This section focuses on the federated part of the architecture.

With the exception of the Data Processing Procedures component, which is not required in the federated architecture, each of the monitoring component has a federated counterpart. The Cloud cluster has been selected for the federated components due to the scalability of its resources.

2.2.1. Federated Data Storage

An additional Prometheus Server is deployed in the Cloud cluster. This Prometheus server aggregates in a single time series database the metrics from all of the clusters.

It offers the same PromQL interface as the previously described for the non-federated Prometheus Server, with the same capabilities. Additionally, a custom API developed with FastAPI has been developed to allow for a simpler interface between the orchestration and the monitoring systems. This API is further described in Section 3.4.

2.2.2. Federated Data Collectors

The Prometheus Server deployed in each cluster as its own Data Storage acts as a Federated Data Collector towards the federated architecture.

2.2.3. Federated Data Transport

The main issue to be solved by the multicluster distributed monitoring architecture is the communication among the Federated Data Storage and Federated Data Collectors as they are placed in separate clusters. The multicluster technologies described in Section 3.3 enable this communication and therefore act as the Federated Data Transport component for the monitoring architecture.

2.2.4. Federated Visualization

Similar how a Grafana instance provides with visualization capabilities to each of the non-federated Data Storage components, another Grafana instance provides the Federated Visualization capabilities required by the system.

2.3. Metrics Implemented

This section summarizes the full set of metrics fully implemented. The last 3 sections, metrics related to power consumption, GPU and Nvidia jetson are new metrics implemented.

2.3.1. CPU

Several CPU related metrics were implemented to monitor and optimize resource usage across the Kubernetes cluster. These metrics enable the identification of performance bottlenecks and support efficient resource allocation:

- **Cluster CPU Usage:** provides a high-level overview of total CPU usage across all cluster nodes as a percentage, enabling quick detection of potential issues.
- **Containers CPU Usage:** reports the CPU consumption per container, considering the number of cores assigned to each.
- **Processes CPU Usage:** monitors CPU utilization by individual processes, showing their total usage related to allocated cores.
- **Pods CPU Usage:** captures the CPU usage per Kubernetes POD, providing insight into workload distribution at the POD level.
- **System Services CPU Usage:** measures the CPU consumption of background system services, supporting infrastructure-level diagnostics.
- **Namespace CPU Usage:** aggregates CPU usage per namespace, offering visibility into resource consumption by application (all components that are part of an application need to be assigned to the same namespace).

These metrics allow for a detailed and aggregated analysis of CPU performance across the cluster.

2.3.2. Memory

The following memory related metrics were implemented to monitor resource usage and support effective capacity planning within the Kubernetes cluster:

- **Cluster Memory Usage:** offers a global view of memory consumption across all nodes, shown as a percentage. It helps quickly detect potential memory bottlenecks at the cluster level.
- **Containers Memory Usage:** provides the amount of main memory used by each container, allowing a detailed analysis of memory distribution across workloads.
- **Processes Memory Usage:** reports memory consumption by individual processes running within the cluster, allowing detailed process-level monitoring.
- **PODs Memory Usage:** reflects the memory usage of each Kubernetes POD, allowing visibility into memory demand at the POD level.
- **System Services Memory Usage:** measures the memory utilization of system services running within the cluster, which helps to measure the overhead of the infrastructure components.
- **Namespace Memory Usage:** aggregates memory usage per namespace, allowing to track memory consumption at application-level (all components that are part of an application need to be assigned to the same namespace).

These metrics enable both high-level monitoring and detailed inspection of memory usage to support efficient resource allocation and system stability.

2.3.3. Networking

A set of network related metrics was implemented to monitor data transfer activity and diagnose potential network-related issues within the Kubernetes cluster:

- **Cluster Network I/O Pressure:** measures the total volume of incoming and outgoing network traffic across the entire cluster, providing a high-level view of network throughput.
- **Containers Network I/O Pressure:** reports the network usage per container, tracking the bytes transmitted and received per second to evaluate communication performance at the container level.
- **Processes Network I/O Pressure:** captures the network usage for individual processes, enabling detailed analysis of process-specific network behavior.
- **PODs Network I/O Pressure:** this metric reflects the data transfer activity of individual PODs running in the cluster, supporting workload-level diagnostics of network performance.
- **Namespace Network I/O Pressure:** aggregates incoming and outgoing traffic by namespace, offering insight into application-level network activity (all related entities need to be grouped under the same namespace).

These metrics enable comprehensive monitoring of network behavior across the different components of the cluster, which helps in optimization and troubleshooting tasks.

2.3.4. Storage

The following storage related metrics were implemented to monitor disk usage and throughput within the cluster, ensuring efficient resource allocation and early detection of capacity constraints:

- **Node Disk Throughput:** measures the read and write rates (in bytes per second) at the node level.
- **Namespace Disk Throughput:** aggregates disk read and write activity per namespace, allowing the monitoring of storage performance for specific applications (all related components need to be deployed under the same namespace).

2.3.5. System

System-level metrics are essential to ensure the cluster can dynamically adapt to changes in infrastructure availability:

Available Compute Nodes: tracks the current number of compute nodes in the cluster, reflecting the system's capacity to run workloads. This metric is crucial for understanding infrastructure scalability and detecting changes due to node failures or additions.

2.3.6. Power consumption

Power consumption and resource utilization at node and container level metrics help optimize energy efficiency in Kubernetes clusters. These insights support energy-aware scheduling, performance tuning, and sustainability reporting:

Energy consumption metrics by container:

- **kepler_container_joules_total:** is the aggregated total energy consumed by CPU, DRAM, GPU, and other components for a given container.
- **kepler_container_core_joules_total**: measures the energy used by CPU cores.
- **kepler_container_dram_joules_total:** describes the total energy used in DRAM by a container.
- **kepler_container_uncore_joules_total:** measures the cumulative energy consumed by uncore components (e.g., cache, memory controller).
- **kepler_container_package_joules_total:** measures the total energy for the CPU socket (cores + uncore).
- **kepler_container_other_joules_total:** measures the cumulative energy consumed by other host components (e.g., SOC, motherboard), often derived from ACPI/IPMI sensors.
- **kepler_container_gpu_joules_total:** measures the total energy consumed by GPUs; currently supports NVIDIA via NVML package.
- **kepler_container_energy_stat** is a compound metric used for model predictions.

Resource Usage Metrics per Container:

- **kepler_container_bpf_cpu_time_us_total:** is the base metric and, it measures the total CPU time used by the container, collected via BPF tracing (always exposed).

Other optional metrics can be enabled related to Hardware and IRQ:

- **Hardware Counter Metrics:**
  - **kepler_container_cpu_cycles_total:** measures the total CPU cycles used; useful for analysing performance, especially in variable-frequency systems.
  - **kepler_container_cpu_instructions_total:** measures the total CPU instructions executed by the container; a standard metric for CPU utilization.
  - **kepler_container_cache_miss_total:** measures the total last-level cache (LLC) misses.
- **IRQ Metrics (via BPF):**
  - **kepler_container_bpf_net_tx_irq_total:** measures the total network packets transmitted by the container.
  - **kepler_container_bpf_net_rx_irq_total:** measures the total network packets received by the container.

- **kepler_container_bpf_block_irq_total:** measures the total block I/O operations initiated by the container.

For the **node level** energy metrics, we have the same structure but aggregated at the node level, e.g. kepler_node_core_joules_total, kepler_node_dram_joules_total, kepler_node_uncore_joules_total, etc.

2.3.7. GPU

Monitoring GPU is essential to ensure optimal performance, cost efficiency and HW reliability. To that end, the NVIDIA GPU operator deploys the dcgm-exporter, which exposes GPU usage metrics such as power draw, memory usage, temperature, and SM utilization. This helps monitor performance, detect hardware issues early, and improve GPU resource efficiency. It also supports GPU cost tracking, workload consolidation, and thermal/power-based scheduling in high-performance environments.

- **GPU utilization** (dcgm_gpu_utilization): measures the percentage of time that the GPU is actively processing workloads over a given sampling interval.
- **GPU Memory:**
  - dcgm_fb_used: measures the frame buffer (GPU memory) usage in bytes.
  - dcgm_fb_total: measures the total GPU memory available in bytes.
  - dcgm_fb_free: measures the free GPU memory in bytes.
- **Power and Energy:**
  - dcgm_power_usage: *measures* the current power draw of the GPU in watts.
  - dcgm_energy_consumption: *measures* the total energy consumed by the GPU in millijoules (some setups).
  - dcgm_power_limit: *measures* the enforced power cap for the GPU in watts.
- **Temperature:**
  - dcgm_gpu_temp: measures the current temperature of the GPU in degrees Celsius.
- **ECC and Errors:**
  - dcgm_xid_errors: this counts NVIDIA XID error events.
  - dcgm_ecc_errors: this counts ECC errors by type (volatile/persistent) and location (l1 cache, l2 cache, etc.).
- **PCIe Metrics:**
  - dcgm_pcie_rx_throughput: measures data received over PCIe in bytes per second.
  - dcgm_pcie_tx_throughput: measures data sent over PCIe in bytes per second.
- **Process-Level Metrics (if enabled):**
  - dcgm_process_gpu_utilization: measures GPU usage per process.
  - dcgm_process_fb_used: measures GPU memory used per process.

2.3.8. NVIDIA jetson

As the previous metrics were not available for NVIDIA jetson HW, a custom exporter was developed to monitor this kind of HW.

- **System uptime:** measures the amount of time the device has been running continuously without a restart, reboot, or shutdown.
- **GPU usage:** measures the percentage of time the Jetson device's GPU is actively processing compute tasks.
- **CPU usage:** measures the percentage of total CPU capacity currently being used by all active processes.
- **Fan:** reports the current speed of the system fan, typically in RPM (revolutions per minute) or as a percentage of maximum speed. Monitoring this helps ensure the cooling system is functioning correctly under load.
- **RAM used:** measures the amount of system memory (RAM) currently in use by applications and processes.
- **Available disk space:** measures the remaining free storage capacity on the Jetson device.
- **Sensor temperatures:** measures the current temperature of various components (e.g., CPU, GPU, PMIC, etc.) via onboard sensors.
- **Power usage:** measures the current power consumption of the Jetson board in watts or milliwatts.

2.4. Validation and Fulfillment of Requirements

This section presents the final validation of the implemented monitoring features and their alignment with the initial project requirements.

As defined in deliverable D3.1, the monitoring system aims to collect, process, store, and report data on the operation, status, and resources of the compute continuum. This data supports the orchestrator and other EXTRACT applications in optimizing performance, availability, scalability, security, and overall system management.

The following table summarizes the defined requirements, and the tools used to address them. Detailed information on each requirement and its relevance to the system can be found in Section 4 of deliverable D3.1.

Monitoring requirements		
Requirement	**Tool used to fulfill the requirement**	**Implementation status**
Near real-time monitoring	Prometheus	Implemented
Flexibility and extensibility	Prometheus + Open Telemetry	Implemented
Integration	API (FastApi + Prometheus client)	Implemented
Historical Data	Prometheus (time series database)	Implemented

Scalability and efficiency	Related to architecture	Implemented
Reliability	Related to architecture	Implemented
Security and compliance	Prometheus	Implemented

Table 2. Monitoring requirements and tools

While the system requirements focus on infrastructure-level monitoring, the metric requirements address agent-level data collection. The table below outlines these requirements along with the tools used to fulfill the requirement. Further details on each requirement and its importance to the system are available in Section 4.2 of deliverable D3.1.

Metric requirements		
Requirement	**Tool used to fulfill the requirement**	**Implementation status**
Metrics related to the nodes and the infrastructure of the compute continuum:		
Available nodes	kube-state-metrics metrics-server	Implemented
CPU usage		Implemented
Memory usage		Implemented
Network Throughput		Implemented
Storage Utilization		Implemented
Node Health	kube-state-metrics	Implemented
Workload Distribution		Not Implemented
Resource Efficiency		Implemented
Metrics related to the applications and systems deployed:		
Application Availability	kube-state-metrics	Implemented
Application Response Time		Not Implemented

Container Metrics	kube-state-metrics metrics-server	Implemented
Network Throughput		Implemented
Storage Utilization		Partially Implemented.
Workload-specific Metrics	kube-state-metrics	Implemented
Service Health		Implemented

Table 3. Metric requirements and tools

3. Data-driven Workflow Deployment and Scheduling

This section presents the final architecture, implementation, and technology integration efforts related to data-driven workflow deployment and scheduling within the EXTRACT platform. The focus is on achieving an efficient, flexible, and scalable orchestration layer that seamlessly integrates heterogeneous computing resources across the compute continuum. The system aims to dynamically allocate workflows while optimizing for latency, throughput, energy consumption, and other relevant performance metrics.

3.1. Orchestration architecture

As stated in D3.2, the EXTRACT platform will be composed of two orchestration layers: the application layer and the infrastructure layer. The application layer schedules the data-driven workflow, and the infrastructure layer manages its deployment and execution.

In the EXTRACT context, the application layer is implemented with COMPSs, that follows a master/worker paradigm. The master creates a Task Dependency Graph (TDG), where nodes are tasks and edges are the data dependencies among them, and then schedules these tasks in the workers. On the other hand, the infrastructure layer is implemented with Kubernetes and is in charge of deploying COMPSs as Pods and ensuring the full workflow execution and ensuring master-workers communication.

It is worth mentioning that both, COMPSs and Kubernetes, have their own scheduler and orchestrator, and it may be easy to confuse the terms. For the sake of the reader's clarity, we propose the next definitions:

- **Application scheduler**: The COMPSs scheduler that decides in which worker task of the TDG will be executed, considering network and data-locality. The default COMPSs scheduler is es.bsc.compss.scheduler.orderstrict.fifo.FifoTS, that prioritizes task scheduling order in FIFO.

- **Application orchestrator**: The COMPSs orchestrator that oversees offloading a task to the decided COMPSs worker.
- **Infrastructure scheduler**: The Kubernetes scheduler that will determine where the COMPSs master and workers Pods are deployed. The default scheduler is kube-sched.
- **Infrastructure orchestrator**: The Kubernetes orchestrator is a broader term that encompasses the entire Kubernetes platform, which is responsible for managing and coordinating the containerized workloads across the continuum.

3.2. Technology Integration and Enhancements

This subsection details the technology integration efforts undertaken to ensure that the orchestrator is seamlessly embedded within the EXTRACT platform, covering Kubernetes integration, COMPSs enhancements, and multicluster capabilities.

3.2.1. Kubernetes Integration

As detailed in D3.2 and the extensive analysis of alternatives depicted in D3.1, the orchestration platform leverages Kubernetes as the primary container orchestration tool due to its maturity, extensibility, and strong ecosystem support. However, significant effort has been dedicated to customizing Kubernetes to fulfill EXTRACT's requirements for extreme data scenarios:

- Kubernetes/COMPSs Connector: A dedicated connector has been developed to enable COMPSs to interact with Kubernetes clusters efficiently. This connector allows COMPSs to submit tasks as Kubernetes Jobs or Pods, managing the execution lifecycle, resource allocation, and dependencies between tasks. It ensures that COMPSs tasks are scheduled with awareness of cluster metrics, such as CPU, memory, and network usage, which are exposed through the monitoring subsystem described in Section 5.
- Multicluster Support: Recognizing the need for workload mobility and resilience across geographically distributed infrastructures, taking full advantage of the compute continuum, the orchestration layer has been extended to support multicluster Kubernetes environments. This integration enables:
    - Dynamic workload migration between clusters.
    - Transparent service discovery across clusters.
    - Integration with tools like Linkerd, Skupper, or Clusterlink (currently under evaluation and further described in Section 3.3) to provide secure, low-latency communication between clusters.
    - The multicluster component also ensures that workloads can adapt to dynamic conditions, such as changing network topologies or resource availability, without disrupting the user experience.

3.2.2. COMPSs

Within the EXTRACT orchestration stack, COMPSs continues to serve as the core parallel programming model, enabling the distributed execution of workflows across the compute continuum. In this final release, substantial enhancements have been introduced to better support dynamic orchestration, particularly in heterogeneous and multi-cluster environments, while providing stronger observability and scalability capabilities.

One of the main advancements is the integration of COMPSs with the monitoring infrastructure, enabling it to retrieve real-time metrics from Prometheus and Pushgateway. A new metrics consumer module has been embedded within the COMPSs runtime, which allows the scheduler to make data-informed decisions about task placement and resource usage. These metrics include system-level information like CPU load and memory availability, as well as application-level metrics such as task execution time. The scheduling algorithm, discussed in more detail in Section 3.4, now supports dynamic task allocation based on these insights, helping to maintain performance targets even under extreme data and compute conditions.

In addition, the system enables direct communication between the running application and the COMPSs scheduler master. This bi-directional exchange allows applications to issue resource hints or signal inefficiencies, which the scheduler can incorporate into its decision-making process.

To enhance observability and elasticity, the COMPSs scheduler also consumes monitoring data from the cluster-wide monitoring API implemented by Ikerlan. This data source provides a comprehensive view of node and container resource usage, enabling the scheduler to make more informed decisions regarding scaling actions.

In parallel, a custom monitoring tool facilitates two-way communication between the application and the COMPSs master, while also linking the runtime with the Prometheus server. This tool acts as an integration layer that enhances observability and provides fine-grained control.

By defining these communication layers between COMPSs, the Kubernetes infrastructure, and the workflow runtime, the scheduler is empowered to take diverse actions concerning where and how many resources should be allocated for workflow execution. These actions include both vertical and horizontal scaling strategies, which will be further detailed in the next section.

The figure below illustrates the monitoring infrastructure deployed within the Kubernetes cluster. Each workflow is encapsulated in a Pod containing the COMPSs Master runtime, Redis, and the internal monitoring tool. This setup integrates with Prometheus via a ServiceMonitor managed by the Prometheus Operator, allowing metrics to be collected and visualized in Grafana.

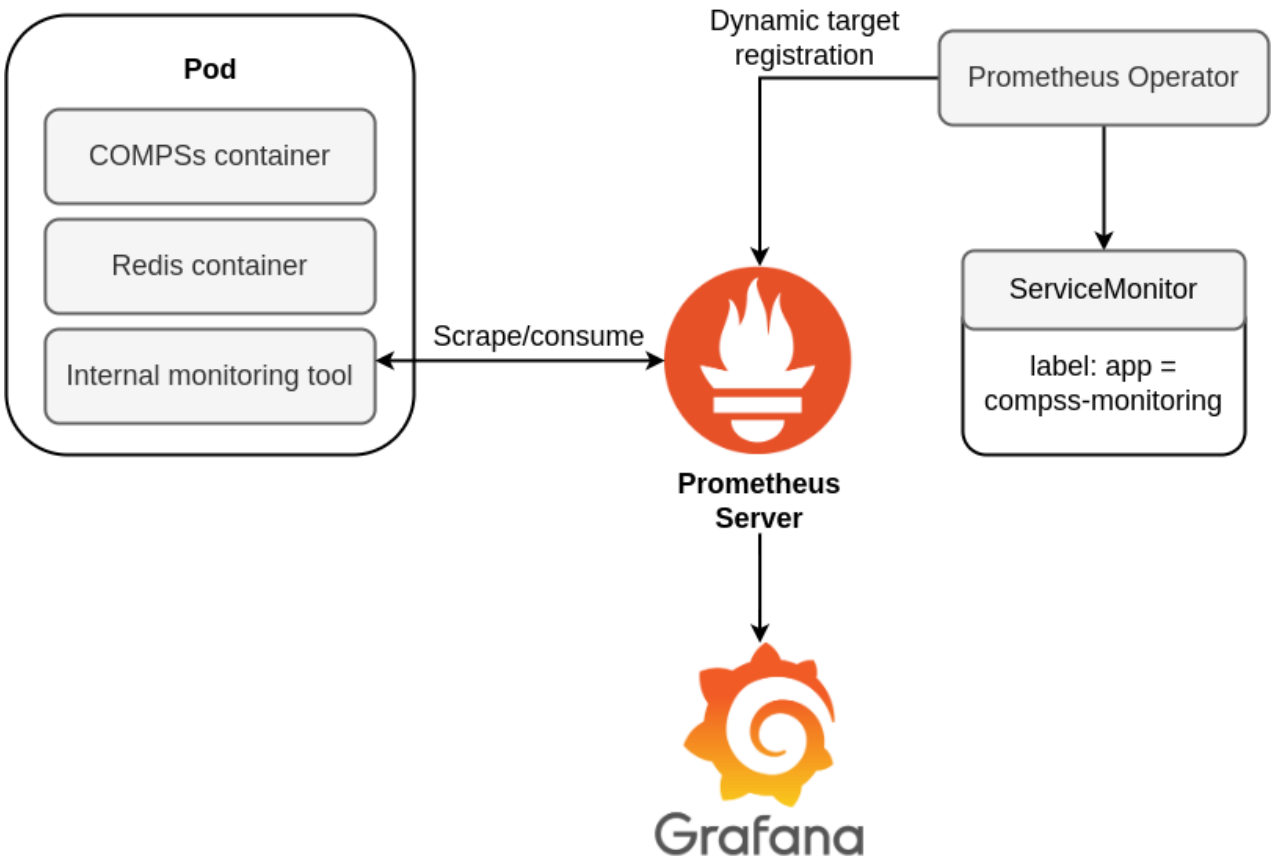

Figure 2: Monitoring Infrastructure

The final release introduces explicit support for both horizontal and vertical scalability, making COMPSs more adaptable to real-world, dynamic workloads.

- **Horizontal scalability** allows the orchestrator to dynamically increase or decrease the number of active worker nodes (Pods) in a Kubernetes cluster, depending on system load and task queue status. This enables rapid response to workload spikes—such as in batch processing or highly parallelizable tasks—and supports resource optimization by scaling down when activity decreases. For example:
  - In the TASKA use case, this is essential to handle large volumes of astronomical datasets, where calibration and imaging tasks can be parallelized efficiently.
  - In PER, during reinforcement learning inference under real-time constraints, horizontal scaling ensures sufficient workers are spun up to reduce inference time and meet safety-critical deadlines.

- **Vertical scalability**, in contrast, refers to adjusting the computational capacity of individual worker nodes. Leveraging Kubernetes resource allocation mechanisms, COMPSs workers can dynamically request additional CPU, memory, or even GPU resources at runtime. This approach is particularly beneficial for computing intensive or non-parallelizable tasks such as model training or signal processing.

  The vertical scalability mechanism in COMPSs follows a multi-phase workflow:
  - **Detection phase:** The system monitors for signs of inefficiency, such as excessive CPU or memory usage, either reported by the application or inferred from observed metrics. If resource saturation is detected, the scheduler may propose increasing the resource allocation for specific containers. Additionally, this phase also verifies whether the workflow is efficiently utilizing the CPUs currently assigned to the worker schedulers. If not, it may reduce the number of assigned CPUs to alleviate vertical scaling pressure, thereby improving overall resource sharing in a multi-tenant environment with multiple concurrent workflows.
  - **Resource fit analysis:** Before proceeding, the system verifies whether the hosting node has enough available capacity to accommodate the increased resource needs.
  - **Evaluation**: Post-adjustment metrics are analysed to determine whether the scaling action was effective. Additional actions may follow if performance targets are not yet met.

Together, these scalability features bring major advantages:

- Better resource efficiency, by matching execution context to workload needs.
- Seamless support for heterogeneous infrastructures, including edge and HPC nodes with different capabilities.
- Faster and more reliable execution for extreme-data applications across domains.

In earlier deliverables, we introduced the runcompss-k8s CLI, a tool to simplify COMPSs deployment in Kubernetes environments. Initially, the master was deployed via a Kubernetes Deployment with one replica and a ClusterIP service, while workers were instantiated using a StatefulSet and a Headless Service to maintain stable DNS identities.

However, this approach posed significant challenges in multi-cluster environments, as most multi-cluster networking tools are designed to handle ClusterIP services, not Headless ones. To overcome this, the deployment model was redesigned:

- The COMPSs master remains deployed using a standard Deployment with a stable ClusterIP service.
- Each COMPSs worker is now deployed as an independent Deployment with one replica, each accompanied by its own ClusterIP service. These services follow a naming convention (application-name-worker-N) so that the COMPSs master can always reconnect to the right worker—even after a Pod restart—ensuring the runtime's internal logic remains stable and predictable.

This change allows seamless integration with multi-cluster tools like Skupper, Linkerd, and Clusterlink, Istio or Submariner (described further below) which rely on service mirroring or renaming conventions. For example, a local service compss-worker-0 might be mirrored to a remote cluster as compss-worker-0-clusterB.

The COMPSs runtime itself has been enhanced to support multi-cluster deployments. It now allows flexible registration of worker nodes based on Kubernetes environment variables, and dynamically adjusts its master-worker communication patterns. Workers can connect to master or master-cluster-name depending on which cluster they belong to, facilitating transparent task orchestration across clusters.

These capabilities are complemented by an improved Kubernetes connector written in Java, which manages dynamic worker lifecycle operations during execution:

- *create*: deploys a new worker and its service
- *waitUntilCreation*: blocks until the worker is ready and registers it
- *createMultiple*: scales up multiple workers in batch
- *destroy and close*: terminate individual or all workers, respectively, once they're no longer needed

Finally, as described in deliverable D1.3, we have added a functionality that allows human interaction: the workflow is halted at some point until it receives a signal via MQTT indicating that it should resume its execution.

3.2.3. Nuvla Data-Catalogue

In the context of the EXTRACT project, the integration focuses exclusively on the data catalogue capabilities of the Nuvla platform, rather than its container orchestration features. This aligns with the project's objectives related to metadata enrichment, cross-provider data discoverability, and lifecycle management of datasets.

As previously described in D3.2, the Nuvla data catalogue is built around three core resources:

- data-object, which acts as a proxy to an S3 object,
- data-record, which stores metadata associated with data-objects and provides an annotation mechanism, and
- data-set, which groups objects and records through filter-based collections.

These components remain central to the data management strategy in EXTRACT, facilitating structured, flexible, and interoperable access to distributed data.

Since D3.2, a notification system has been developed and integrated into the Nuvla data catalogue. This new feature enables real-time notifications via MQTT whenever a new S3 object is published to Nuvla. Specifically, once a new data-object is created and made publicly accessible, a message is published to a designated MQTT topic. This allows subscribed applications or services to react immediately to the presence of new data — enabling event-driven workflows and improving responsiveness in downstream processing pipelines.

The notification system introduces an important mechanism for decoupled, asynchronous communication within the data ingestion and utilization workflows. It enhances the ability of EXTRACT components and third-party services to stay up-to-date with the availability of new data, supporting more dynamic and automated data ecosystems.

3.3. Multicluster Technologies

In the EXTRACT platform, multicluster technologies are a foundational component that enable the abstraction and interconnection of distributed computational resources across the compute continuum. These technologies make it possible to treat a set of physically separated Kubernetes clusters—running on the edge, in the cloud, or in HPC centers—as a coherent, logically unified execution substrate.

This abstraction is not just a convenience; it's essential to fulfill the project's goal of orchestrating extreme-data workflows that dynamically adapt to changing workloads and infrastructure conditions. In real-world use cases like TASKA or PER, data is inherently distributed and dynamic. For instance, large volumes of radio astronomy data may be collected at remote sites such as Nançay Radioastronomy Observatory, while video or positional data in a smart city context may arrive from edge devices in Venice. Efficient processing of these data streams requires deploying parts of the application stack in proximity to data sources, and other parts in high-performance compute environments. This is only feasible if clusters can interact seamlessly and securely.

Without robust multicluster mechanisms, such flexible deployment and dynamic adaptation would be infeasible. Clusters would operate as silos, and transferring data or services between them would require manual intervention, complex VPN setups, or static interconnects—each of which is error-prone, brittle, and not scalable.

The heterogeneity of computing environments in EXTRACT—ranging from low-latency edge clusters to high-throughput cloud backends—means that different components of a data processing pipeline are best executed in different locations. A monolithic, single-cluster model is insufficient for handling these needs:

- In TASKA, for example, image reconstruction from interferometric data benefits from edge data ingestion and preprocessing, while compute-intensive calibration is delegated to HPC clusters.
- In PER, reinforcement learning inference must occur near real-time at the edge, but model training requires massive parallelism only available on powerful cloud or HPC backends.

Moreover, resilience and scalability are key requirements. When clusters are interconnected, EXTRACT gains the ability to reroute or failover workloads, mitigating bottlenecks or outages. It also allows optimal resource usage by selecting the most cost- or energy-efficient site for a given task at runtime. This is particularly important in workflows with unpredictable load spikes or time-sensitive deadlines.

Multicluster approaches for Kubernetes-based platforms typically fall into one of three architectural categories:

- **Pod Offloading (Virtual Kubelets):** The Kubernetes control plane in one cluster is extended to "see" nodes in another cluster via virtual nodes. This enables it to schedule Pods remotely, treating foreign resources as local.
- **Service Mirroring (Service Exporting):** Services in one cluster are mirrored in another as proxy endpoints, allowing cross-cluster service discovery and communication without replicating full workloads.
- **Control Plane Federation:** A central control plane coordinates and synchronizes resource definitions (e.g., Deployments, ConfigMaps) across multiple clusters, often using policies for propagation.

For EXTRACT, after thorough analysis, the service mirroring model was selected. This approach is better aligned with the design of COMPSs, which maintains a master-worker model and benefits from being able to register remote workers as local services. With mirrored services acting as proxies, the COMPSs master can offload tasks transparently, without incurring the overhead of full Pod replication or exposing internal runtime complexity.

The multicluster technology selected for EXTRACT must satisfy several critical requirements:

- Low-latency, secure service-to-service communication across clusters.
- Seamless integration with Kubernetes networking and service discovery.
- Support for observability and traffic routing policies.
- Compatibility with COMPSs' dynamic scheduling and task offloading model.
- Minimal administrative overhead and ability to support scaling across more than two clusters.

To meet these needs, we identified five promising technologies for evaluation: Clusterlink, Skupper, Linkerd, Istio and Submariner. These solutions are currently being tested and benchmarked under realistic scenarios, focusing on factors such as failover handling, throughput, integration complexity, and security posture. Although we are at Month 30 (M30) of the project, most performance evaluation will occur during the final phase (M30–M36).

1. **Clusterlink** [1] ClusterLink simplifies the connection between services that are located in different domains, networks, and cloud infrastructures. ClusterLink runs on Kubernetes-enabled clusters. The ClusterLink deployment operator allows easy deployment of ClusterLink to a Kubernetes cluster.

ClusterLink setup operations (create, deploy, delete) are performed using the ClusterLink CLI command. Once the ClusterLink operator is deployed on a cluster, operations (peer, export, import, policies) are performed through Custom Resource Definition objects (CRDs). The creation and manipulation of CRDs is done through the Kubernetes API. Examples of the CRD formats for these operations can be found at https://clusterlink.net/docs/main/tasks/operator/

The ClusterLink gateway contains the following internal subcomponents:

- Control Plane (CP) is responsible for maintaining the internal state of the gateway, for all the communications with the remote peer gateways by means of the ClusterLink CP Protocol, and for configuring the local data plane to forward user traffic according to policies. Part of the control plane is the policy engine that can also apply network policies (ACL, load-balancing, etc.)
- Data Plane (DP) responds to user connection requests, both local and remote, initiates policy resolution in the CP, and maintains the established connections. ClusterLink DP relies upon standard protocols and avoids redundant encapsulations, presenting itself as a Kubernetes service inside the cluster and as a regular HTTP endpoint from outside the cluster, requiring only a single open port (HTTP/443) and leveraging HTTP endpoints for connection multiplexing.

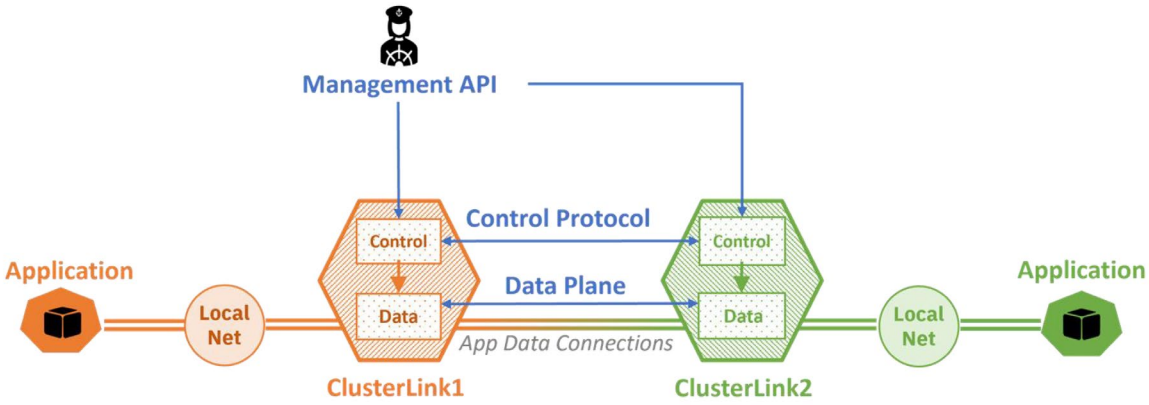

Figure 3: ClusterLink Architecture

ClusterLink leverages the Kubernetes API using CRDs to configure cross-cluster communication. ClusterLink management is based on the following key concepts:

- Peers: Represent remote ClusterLink gateways and contain the metadata necessary for creating

protected connections to these remote peers.

- Exported services: Represent application services hosted in the local cluster and exposed to remote

ClusterLink gateways as Imported Service entities in those peers.

- Imported services: Represent remote application services that the gateway makes available locally

to clients inside its cluster.

- Policies: Represent communication rules that must be enforced for all cross-cluster communications at each ClusterLink gateway.

For further information, please refer to the concepts section on the ClusterLink website: https://clusterlink.net/docs/latest/concepts/.

2. **Skupper** [2] introduces a virtual application network between clusters, operating primarily at the application layer (L7). It enables secure communication between services in different Kubernetes clusters without requiring changes to the underlying network infrastructure or Kubernetes itself. This makes it especially appealing in constrained or security-sensitive environments where modifying cluster networking (e.g., with VPNs or VPC peering) is impractical.

Skupper uses AMQP internally to route traffic between clusters and leverages mTLS to encrypt traffic. Its developer-friendly model and minimal configuration make it ideal for rapid integration—an important factor in the PER use case, where responsiveness and deployment flexibility are essential.

However, since it introduces application-level proxies, we are currently evaluating its performance under high-throughput workloads, as well as the potential impact on latency-sensitive inference steps at the edge.

3. Linkerd [3] is a mature, widely adopted service mesh that supports transparent multicluster operation through its built-in extensions. Unlike the previous two, Linkerd provides layer 4 and 7 observability, traffic policies, retries, and security out of the box, with seamless Kubernetes integration.

This combination of observability, policy control, and performance makes Linkerd a strong candidate for EXTRACT. Its multi-tenant support and dynamic configuration options allow it to scale to complex orchestration scenarios across edge, cloud, and HPC.

The trade-off lies in operational complexity: deploying and maintaining Linkerd in a multicluster topology requires deep integration with CI/CD pipelines, security policies, and monitoring infrastructure. Our evaluation is focusing on whether these operational demands are justified by the robustness and flexibility it provides for orchestrating workflows across sites.

4. **Istio** [4] is one of the most comprehensive and feature-rich service meshes currently available, and it includes robust multicluster capabilities. Built around the Envoy proxy, Istio provides extensive support for traffic management, security, observability, and policy enforcement at both L4 and L7. In a multicluster setting, Istio enables service discovery and communication between clusters through service mirroring, supported by the Istio control plane (Istiod), which maintains a consistent view of all registered services across clusters.

For EXTRACT, Istio offers advanced capabilities that could be especially relevant as the orchestration complexity increases—for example, dynamic routing rules, circuit breaking, and telemetry aggregation across environments. This level of control may benefit use cases like PER, where routing policies need to be adapted in near real-time to accommodate evolving evacuation scenarios or fluctuating user demand.

However, the complexity of Istio's setup and maintenance, particularly when extending to multicluster topologies, poses significant challenges. It introduces additional resource overhead, and managing the control plane components across administrative domains can become cumbersome. Therefore, we will carefully evaluate whether the benefits in terms of observability and policy granularity justify the operational cost within the EXTRACT orchestration stack.

5. Submariner [5] offers a network-layer approach to multicluster connectivity, with a primary focus on enabling direct pod-to-pod communication across Kubernetes clusters. It creates secure tunnels between clusters using strongSwan (for IPsec) or WireGuard and sets up custom routing tables to make remote Pods appear as if they were on the same network. This is particularly attractive in low-level, latency-sensitive scenarios where transparency and native IP reachability are critical.

Unlike solutions like Skupper or Istio, Submariner operates below the service level, which makes it well-suited for tightly coupled applications or custom runtimes like COMPSs that benefit from low-level access to remote nodes.

Submariner could be useful for use cases such as TASKA, where bulk data transfers between compute nodes and storage clusters must happen with minimal overhead. However, since Submariner manipulates cluster-level network configurations and routing tables, it may require elevated privileges and careful security considerations. Moreover, its support for advanced features like service-level policy enforcement or application-layer telemetry is limited compared to higher-level meshes, which may constrain its applicability in more complex orchestration scenarios.

As of M30, EXTRACT is entering its final evaluation phase. While multicluster connectivity is functional in our current deployments, we aim to benchmark each technology more extensively during M30–M36, testing:

- Performance under simulated edge-cloud-HPC workloads.
- Integration with COMPSs runtime and orchestration stack.
- Behavior under node failures, cluster isolation, and service recovery.
- Overhead in control plane synchronization and proxying.
- Security guarantees, including mTLS and RBAC enforcement across clusters.

Our final goal is to select a multicluster solution that aligns with EXTRACT's vision of a resilient, secure, and adaptive orchestration layer over the compute continuum—ensuring optimal workload placement, reduced latency, and minimized operational burden.

3.4. Final Integration with Monitoring Platform

According to previous deliverables, D3.1 and D3.2, the main objective of the monitoring platform is to collect, process, store, and report information related to the operation, status, and resources of the compute continuum. This information serves as a critical input for the orchestrator and other EXTRACT applications, enabling them to enhance performance, availability, scalability, security, and overall management of both the compute continuum and the applications deployed across it.

As outlined in the previous deliverable D3.2, the only remaining component to be implemented was the REST API enabling access to the Prometheus time-series database to retrieve the monitoring metrics. This API has now been fully developed using Python, leveraging the FastAPI [6] framework for building efficient, asynchronous web services, and the official Prometheus Python client library to query and process the metrics stored in the time-series database. This implementation ensures seamless integration with the overall monitoring architecture and provides a standardized, extensible interface for other components of the EXTRACT ecosystem—such as the orchestrator or analytical tools—to access real-time and historical operational data in a reliable and performant manner.

The endpoints developed for retrieving metrics are designed to receive POST requests with a JSON payload containing key parameters: the time range for the query, the interval between data points, and the time window used to calculate the rate of change for the selected metric. This design allows clients to perform flexible and fine-grained time-series queries tailored to their specific monitoring or analysis needs.

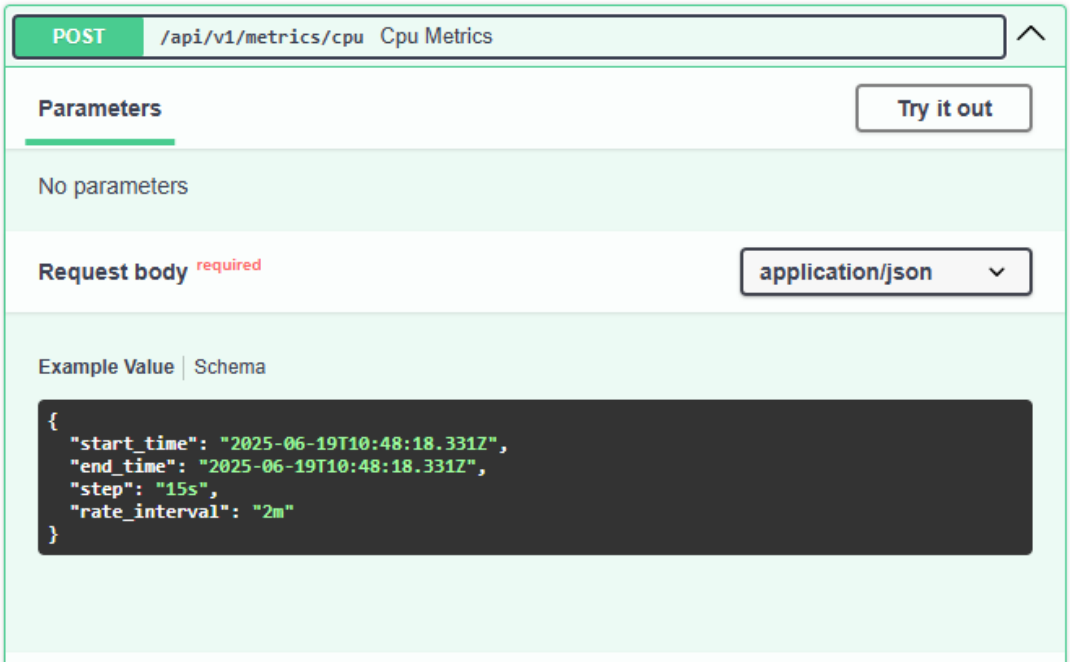

Figure 4: POST request for retrieving CPU consumption metrics

Additionally, for each available metric, there is a dedicated endpoint that allows the retrieval of the latest stored value. This functionality is exposed through a GET request, which accepts the service name as a query parameter to specify the target service for which the metric should be retrieved.

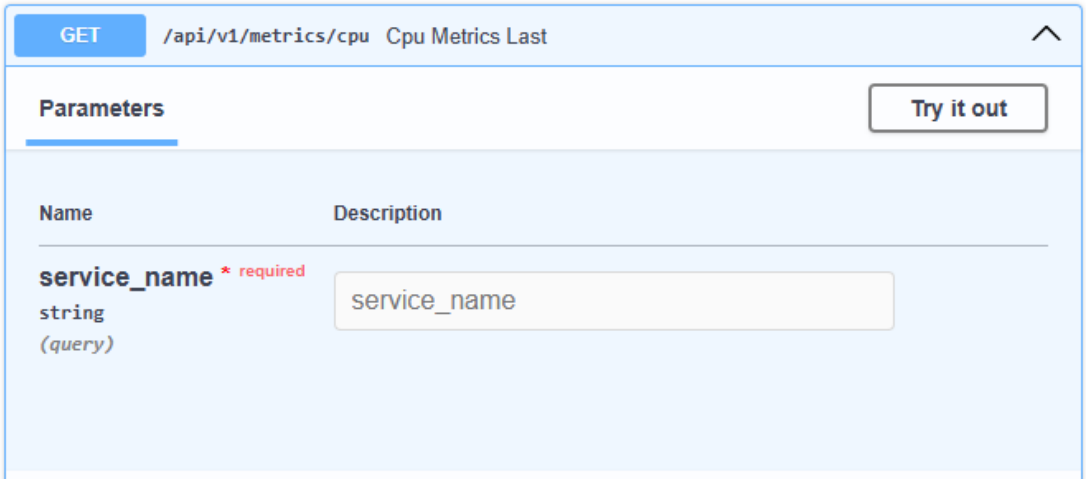

Figure 5: GET request for retrieving the last CPU consumption metric

By offering access to both historical data and up-to-date metric values, the API provides a flexible and efficient interface to support diverse monitoring and operational needs across the compute continuum.

The endpoints are aligned with Prometheus's query model and internally translate the client-specified parameters into PromQL expressions. The returned data is structured in a consistent and easily consumable JSON format, facilitating integration with various components within the EXTRACT ecosystem, such as COMPSs. As shown in the following screenshot, the output returned by the API consists of a list of results grouped by node, where each entry contains a series of timestamped metric values. Each item in the list represents a data point collected from a specific node, including the exact timestamp and the corresponding metric value at that point in time.

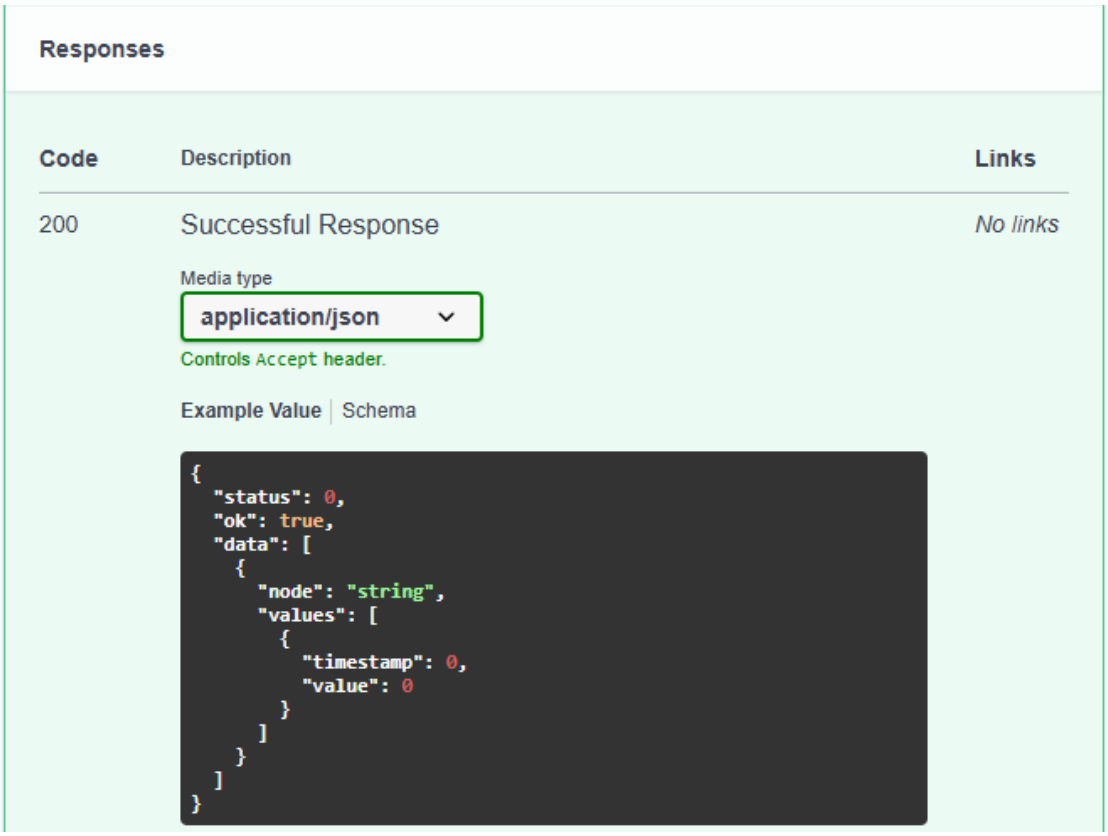

Figure 6: API response structure for all endpoints

In terms of deployment, the monitoring API is containerized using Docker and included as part of the monitoring stack within a Kubernetes-based environment to ensure portability, scalability, and consistency across different infrastructure configurations.

4. Evaluation and Performance Analysis

4.1. Evaluated Tools and Comparison

The increasing reliance on distributed deployments across the compute continuum has made multicluster orchestration a critical capability within the EXTRACT platform. To ensure that workflows can be deployed seamlessly, securely, and efficiently across heterogeneous resources, we are conducting a comprehensive evaluation of leading multicluster technologies. The goal is to assess their performance, operational characteristics, and compatibility with EXTRACT's dynamic orchestration stack, particularly the COMPSs runtime and monitoring components.

To this end, we have selected five multicluster solutions for detailed benchmarking as already presented in the previous section.

These tools reflect a representative spectrum of the multicluster design space:

- **Skupper** implements service mirroring via application-level tunneling, requiring minimal changes to the underlying infrastructure.
- **Linkerd** and **Istio** are full-fledged service meshes, offering observability, security, and traffic management, with built-in multicluster extensions.
- **Submariner** enables IP-level routing between clusters, establishing direct pod-to-pod communication across geographically distributed sites.
- **Clusterlink** is a lightweight, layer 4 communication enabler focused on performance and scalability for cross-cluster service exposure.

So, by including both mesh-based and routing-based approaches, we aim to understand how well each strategy maps to the EXTRACT execution model, and which tool can serve as a stable and performant backbone for orchestrated workflows that span multiple geographical domains.

Each of these tools implements a different approach to enabling multicluster communication, and comes with trade-offs in terms of complexity, performance, scalability, and ease of integration. For EXTRACT, this diversity is especially important since the 2 use cases of the project place very different demands on the orchestration system—ranging from real-time, latency-sensitive control at the edge to large-scale, bandwidth-heavy data ingestion and processing.

The evaluation is being carried out using a custom-developed benchmarking tool, which is based on iperf, a widely used utility for measuring TCP and UDP network performance. To facilitate integration and automation across our Kubernetes-based deployments, a wrapper application was developed by one of the partners (IKERLAN), enabling the deployment and control of the benchmarking process via Ansible.

This tool features a FastAPI-based interface, exposing several endpoints for launching, monitoring, and retrieving results from the iperf tests. The results of each test are exported in CSV format, which supports structured post-processing, visualization, and reproducibility.

The following metrics will be gathered systematically for each solution:

- **Bandwidth** (using TCP): This metric reflects the raw throughput achieved between services located in different clusters, helping assess the ability to support high-throughput applications.
- **Packet Loss Rate** (using UDP): UDP packet loss is a strong indicator of network unreliability or instability, particularly relevant for time-sensitive data flows such as video or streaming telemetry.
- **Latency** (using UDP): Round-trip latency impacts the performance of short-lived or interactive tasks, such as service discovery, API calls, or distributed control loops.
- **Ease of Deployment and Integration**: While not directly quantitative, deployment complexity and compatibility with existing EXTRACT tooling will be documented through setup efforts and qualitative assessment from engineers.

- **Automation and Observability**: The extent to which each tool can be monitored, configured, and scaled using automated infrastructure (e.g., Helm charts, custom operators, monitoring endpoints) is being evaluated as a proxy for operational overhead.

While the benchmarking is still ongoing, the analysis will cover several test scenarios:

1) Short-lived burst communication, relevant for microservice orchestration and dynamic scheduling.
2) Sustained high-throughput streaming, relevant for data-intensive HPC and edge ingestion scenarios.
3) Mixed-mode workloads, where data must be shared bidirectionally between clusters with varying workloads and network topologies.

The results, will be used to inform the final selection of the multicluster backend for EXTRACT during the project's last phase. Our target is to strike a balance between raw performance, ease of use, and operational fit with the rest of the platform components.

4.2. Experimental Setup

To rigorously assess the performance, reliability, and suitability of the selected multicluster technologies within the context of EXTRACT, a diverse and realistic experimental setup has been designed and is currently being deployed incrementally. This infrastructure not only reflects the types of environments where EXTRACT could operate in the use cases, but also allows us to isolate key variables such as geographic latency, administrative isolation, and heterogeneous networking configurations.

At this stage of the project, our deployment consists of three active clusters, with a fourth one scheduled to be added in the coming months:

- A cloud-based cluster hosted on OVH
- Two on-premises clusters deployed at IKERLAN's facilities, which are co-located and interconnected
- An upcoming cluster at BSC's DMZ, expected to become operational during the final evaluation phase (M30–M36)

This topology provides a valuable balance between real-world deployment conditions and controlled testing environments. The OVH cluster introduces realistic cloud latencies and public infrastructure variability, while the IKERLAN clusters allow us to conduct more deterministic measurements, eliminating or minimizing the effect of wide-area network variability. The presence of two local clusters within IKERLAN is particularly useful for differential benchmarking: it enables the team to evaluate the overhead and behavior of the multicluster solutions in a setting where networking conditions are expected to be nearly ideal. This helps isolate the intrinsic overhead introduced by each technology itself, rather than confounding it with external latency or packet loss.

For example, by comparing the latency and throughput observed between the two IKERLAN clusters to those observed between IKERLAN and OVH, we can better understand how tools like Skupper or Clusterlink scale and adapt to distance, congestion, or routing constraints. This layered evaluation strategy is essential to ensure that our selection of multicluster tools is informed not only by best-case scenarios, but also by edge cases that mimic production usage patterns.

The cloud cluster on OVH is provisioned using Kubernetes, with an S3-compatible object store (MinIO) and monitoring infrastructure (Prometheus and Grafana) integrated into the stack. The network topology, access control, and deployment policies in this cluster emulate what a commercial or federated partner in a real-world deployment might provide. This makes it a suitable environment to test how multicluster communication performs when crossing organizational or network boundaries. It also serves as a proving ground for automated provisioning tools such as Ansible and Helm, which are critical in ensuring repeatability of tests and reproducibility of results.

The BSC cluster, to be added shortly, will represent a third environment type: a semi-isolated datacenter network, with hardened access layers and more stringent security controls. Including this cluster will allow us to test how each multicluster tool integrates into environments with network segmentation, NAT traversal, and potential egress restrictions—scenarios commonly encountered when combining HPC, cloud, and edge environments under a single orchestration umbrella.

All clusters are equipped with our custom FastAPI-based wrapper around iperf, which was described in Section 4.1. This tool is deployed as a Kubernetes-native service, enabling us to spin up benchmarking endpoints programmatically via API or through declarative Ansible playbooks. Each cluster exposes this service both internally and externally, allowing inter-cluster metrics to be collected without requiring SSH access or manual test coordination. Additionally, health-check APIs ensure that only healthy nodes are included in measurement sessions, and all test outputs are aggregated in a structured CSV format for offline analysis and visualization.

Tests are run in both directed and reciprocal modes. That is, for each pair of clusters, traffic is initiated in both directions to account for asymmetric network conditions. This bidirectional testing also helps validate the routing behavior and resilience mechanisms of the multicluster tools—e.g., what happens when a mirrored service in one direction fails, or how failover and reconnect logic behave in degraded conditions.

In terms of deployment reproducibility, all clusters share a common configuration baseline, including a standard Kubernetes version (currently v1.28), CNI plugin (Calico), and shared Helm charts for installing the multicluster tools under evaluation. However, local differences in network performance, hardware, and virtualization technologies are intentionally preserved, as they reflect the heterogeneous nature of real-world compute continuum environments. These differences will be carefully documented to help contextualize any performance anomalies observed during testing.

Finally, monitoring and logging are centralized using Prometheus, with each cluster exposing metrics to a common Grafana instance when allowed. This monitoring fabric not only supports the collection of iperf-based performance data but also captures the behavior of the multicluster tools themselves, such as connection health, service discovery delays, and proxy performance.

4.3. Results and Analysis

The evaluation of multicluster networking solutions in EXTRACT is a crucial step to determine their practical viability, performance, and compatibility with the platform's orchestration and monitoring components. This assessment focuses not only on raw performance metrics such as latency, throughput, and packet loss, but also on operational criteria like deployment complexity, observability, resilience, and integration effort.

However, as of the current milestone (M30), the testing phase has not yet begun. The infrastructure and supporting tooling are now in place, and initial deployments have been completed, but the actual benchmarking activities and the subsequent analysis of results are scheduled for the final phase of the project. This last phase, spanning from M30 to M36, will include extensive testing, stress scenarios, and performance validations to ensure that the selected multicluster solution meets the demanding requirements of EXTRACT's use cases.

The comprehensive results, including comparative performance charts, technical trade-offs, and integration challenges observed during the tests, will be fully presented in the final deliverable. These results will inform the final selection of the multicluster component and help consolidate the orchestration architecture that will power EXTRACT's end-to-end workflow execution across the compute continuum.

5. Conclusion

This deliverable summarizes the final release of the development and integration of the EXTRACT orchestration and monitoring components, as part of the project's vision to enable dynamic, secure, and efficient data-driven workflows across the compute continuum. Over the course of this reporting period, we have finalized and stabilized the core orchestration architecture and significantly enhanced the interoperability, adaptability, and observability of our system.

On the orchestration side, the COMPSs runtime has evolved beyond its initial capabilities to support both vertical and horizontal scalability in Kubernetes-based deployments. This dual scalability is fundamental to handling diverse workload characteristics, from massively parallel inference tasks at the edge to compute-intensive model training in the cloud or HPC environments. The inclusion of real-time metric-driven scheduling mechanisms, seamlessly integrated through Prometheus, strengthens our ability to meet stringent QoS demands under dynamic data and resource conditions.

Additionally, we have advanced our work on multi-cluster orchestration. Building upon the original single-cluster design, the current architecture now supports seamless task offloading across heterogeneous Kubernetes clusters using service mirroring approaches. In particular, we have evaluated several multicluster technologies—Clusterlink, Skupper, Linkerd, Istio, and Submariner—assessing their compatibility with COMPSs' programming model and orchestration logic. This is a crucial step toward unlocking the full potential of the compute continuum, enabling workloads to span edge, cloud, and HPC sites without manual intervention.

In parallel, we have made progress on the distributed monitoring infrastructure. This includes the development of metrics collection mechanisms adapted to multi-cluster deployments, and the initial integration of custom performance probes to assess inter-cluster behavior and network overhead. A benchmark campaign has been defined and instrumented, with its results to be published in the next project phase (M30–M36). These evaluations will guide the final selection of the multicluster backend and help validate the system's robustness and adaptability at scale.

The developments reported in this deliverable demonstrate a mature and coherent orchestration and monitoring architecture, tightly aligned with EXTRACT's objectives. While much of the validation and experimentation will occur in the final phase of the project, the building blocks are now in place. This paves the way for full-scale testing with real use cases, allowing us to confirm the scalability, reliability, and performance of the EXTRACT platform under realistic conditions. The coming months will therefore focus on optimization, hardening, and final benchmarking to ensure the platform is ready for deployment in production-like environments.

6. Acronyms and Abbreviations

- BPF – Berkeley Packet Filter
- DMZ – DeMilitarized Zone
- DX.Y – Deliverable X.Y
- IRQ – Interrupt Request
- K8 – Kubernetes
- QoS – Quality of Service
- TX.Y – Task X.Y
- WP – Work Package

7. References

[1] https://clusterlink.net/

[2] https://skupper.io/

[3] https://linkerd.io/

[4] https://istio.io/

[5] https://submariner.io/

[6] https://fastapi.tiangolo.com/