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

D3.4 Data-driven orchestration validation

Version 1.0

Documentation Information

Contract Number	**101093110**
Project Website	www.extract-project.eu
Contractual Deadline	M39, 31 March 2026
Dissemination Level	Public
Nature	Report
Authors	Alba Cañete, Eduardo Quiñones (BSC)
Contributors	IKERLAN
Reviewer	Agathe Veillon (SIXSQ)
DOI	10.5281/zenodo.19208104
Keywords	Data, orchestration, monitoring, scheduling, workflow

Change Log

Version	Description Change
V0.1	Initial draft of the table of contents and collaboration assignments
V0.2	Partners contributions
V0.3	Internal review – Agathe Veillon (SIXSQ)
V1.0	Final release. 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 Deliverable D3.4 – Data-driven orchestration validation, the validation of the work developed in the context of Work Package 3 of the EXTRACT project. WP3 is devoted to the development of data-driven orchestration mechanisms and a distributed monitoring architecture able to support extreme data processing across the compute continuum, from edge to cloud and HPC resources. Building on the requirements and architectural definitions in D3.1, the first implementation reported in D3.2, and the final architecture and integration described in D3.3, this deliverable focuses on the validation of the resulting orchestration and monitoring stack in realistic conditions.

The validation activities reported here are carried out in the context of Task 3.4 – Data-driven orchestration validation (M31–M36), in close collaboration with Task 1.4 and the use case owners. They rely on end-to-end executions of the EXTRACT workflows in the two project use cases, TASKA and PER, using the final WP3 components as described in previous deliverables. The overall objective is to demonstrate that the EXTRACT orchestrator and monitoring infrastructure can select and manage the most appropriate computing resources, while fulfilling the extreme data characteristics specified by the users and keeping the monitoring overhead within acceptable bounds.

1.1. Purpose and scope of the deliverable

The main scope of D3.4 is to validate the data-driven orchestration and monitoring framework developed in WP3, rather than to introduce new architectural elements. In particular, this deliverable:

- Assesses the functional correctness of the EXTRACT orchestrator and monitoring components when executing complete data-mining workflows in the TASKA and PER use cases.
- Evaluates to what extent the orchestration decisions select the most appropriate combination of edge, cloud and HPC resources, taking into account security, performance and energy constraints.
- Analyses how the monitoring system supports those decisions by providing timely and relevant metrics, while preserving confidentiality and limiting overhead.

The validation combines qualitative and quantitative evidence. On the one hand, we verify that the implemented mechanisms behave as expected with respect to the requirements identified in D3.1 and the design choices described in D3.2 and D3.3. On the other hand, we present representative measurements of performance, resource utilisation and energy consumption collected during end-to-end executions in each use case, so that the impact of the orchestration and monitoring choices can be assessed.

Overall, the deliverable aims to provide a clear and traceable argument showing that the WP3 outcomes are mature and effective enough to support extreme data workflows in realistic environments.

1.2. WP3 Objectives and link to Objective O2

WP3 directly contributes to Objective O2 by providing data-driven orchestration and monitoring mechanisms that deploy and execute data-mining workflows on the most appropriate resources across the compute continuum under user-defined extreme data constraints. Within this, O2.1 concerns the deployment and scheduling techniques that choose secure, privacy-preserving and energy-efficient combinations of edge, cloud and HPC resources, while O2.2 focuses on a distributed monitoring architecture that securely observes workflow execution in terms of resources, energy, QoS and security, and supplies this information to the orchestrator. D3.4 is the main deliverable demonstrating that O2.1 and O2.2 have been achieved, through evaluation against KPI2.1, which measures how efficiently and securely extreme data is processed and how energy–performance trade-offs are managed when moving from independent to holistic consideration of data characteristics, and KPI2.2, which measures the monitoring system's ability to provide the required information with minimal (<5%) performance overhead and without exposing sensitive workflow details.

1.3. 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.
D2.4	T2.3	D3.4 presents integration with the data-mining framework evaluated in D2.4 and provides orchestration and monitoring capabilities. In that deliverable, we validate how COMPSs integrates Lithops jobs together.
D4.3	T4.2, T4.3	This deliverable validates the implementation 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.
D4.5	T4.5	Deliverable D4.5 provides a guideline for deploying and executing the Orchestration layer validated in this deliverable.

Table 1. Relationship with other WPs

The validation reported in D3.4 is tightly connected to several other work packages and deliverables of EXTRACT. Within WP3 itself, it builds directly on:

Deliverable	Tasks	Relation
D3.1	T3.1	Data-driven orchestration requirements, which established the initial requirements for orchestration, deployment and monitoring across the compute continuum.
D3.2	T3.2, T3.3	Data-driven orchestration and monitoring (first release), which introduced the first implementation of the monitoring architecture and the initial integration with the orchestrator
D3.3	T3.2, T3.3	Data-driven orchestration and monitoring (final release), which delivered the final architecture and integration of the orchestration and monitoring stack, including federated monitoring and enhanced scheduling mechanisms.

Beyond WP3, the validation relies on the use case definitions and workflows from WP1 and WP2, as well as on the compute continuum abstraction and platform services provided by WP4. The TASKA and PER workflows, described and refined in their respective deliverables, provide the end-to-end scenarios in which the WP3 components are exercised. The compute infrastructures and interoperability mechanisms from WP4 enable the deployment of these workflows across edge, cloud and HPC environments. In addition, Task 1.4 defines the overall validation strategy for the project, to which this deliverable contributes by supplying the evidence related to Objective O2.

1.3.1. Structure of the document

The remainder of the document is organised as follows. Section 2 gives a concise overview of the main WP3 outcomes and the components that are subject to validation, namely the data-driven orchestrator and the distributed monitoring architecture, and clarifies the assumptions and limitations considered. Section 3 describes the validation methodology. Sections 4 and 5 present the validation results for the two project use cases. Each section introduces the corresponding workflows and characteristics, explains how orchestration and monitoring have been validated, and analyses the results in terms of performance, energy, QoS and security. For each use case, the fulfilment of KPI2.1 and KPI2.2 is explicitly discussed in Section 6. Finally, Section 7 summarises the main conclusions and impact of the validation work and outlines remaining challenges and possible directions beyond the project. Sections 8 and 9 list acronyms and references, respectively.

1.4. Role of WP3 Components in the EXTRACT Use Cases

The orchestration and monitoring stack developed in WP3 is not an isolated component, but a core part of the end-to-end EXTRACT platform. Its effectiveness is ultimately assessed in the context of the two project use cases, TASKA and PER, which define concrete extreme data scenarios and workflows.

In both use cases, the data-mining workflows are expressed using the EXTRACT data-mining framework from WP2 and instantiated as COMPSs workflows. The orchestrator, in combination with Kubernetes, is responsible for deploying and scheduling these workflows across the available edge, cloud and HPC resources, taking into account the specific constraints of each scenario. The monitoring architecture continuously observes the execution, providing the orchestrator with the metrics needed to choose the most appropriate computing elements, to balance performance and energy consumption, and to respect security and QoS requirements.

In the TASKA use case, this translates into managing data-intensive processing pipelines over large astronomical datasets, where some stages benefit from being close to data sources and others require high-performance computing capabilities. In the PER use case, the stack supports workflows involving real-time or near real-time analytics and decision support, where low-latency response and energy-aware use of heterogeneous resources are essential. In both cases, the combination of the COMPSs/Kubernetes orchestrator and the federated monitoring system is what enables EXTRACT to implement the data-driven orchestration mechanisms targeted by Objective O2. The remainder of this deliverable details how this stack behaves in practice and to what extent it fulfils the requirements and KPIs defined for WP3.

2. Overview of the EXTRACT Orchestration and Monitoring Stack

This section provides a concise overview of the orchestration and monitoring stack developed in WP3 and validated in this deliverable. It recalls the main components and their roles in the EXTRACT architecture, as introduced and progressively refined in D3.1, D3.2 and D3.3. The detailed architecture and implementation aspects are not repeated here, but summarised to give the reader the necessary context for the validation activities presented in the following sections.

2.1. Distributed Monitoring Architecture

Complementing the orchestrator, WP3 has developed a distributed monitoring architecture capable of collecting, aggregating and exposing metrics from the EXTRACT platform across edge, cloud and HPC environments. The initial requirements and architecture were defined in D3.1, a first implementation was delivered in D3.2, and the final federated version was presented in D3.3.

In each infrastructure cluster, monitoring is built around Prometheus as the central time-series database and scraping engine. A set of exporters, including kube-state-metrics and metrics-server, gathers metrics about nodes, Pods, containers and system services from Kubernetes.

Where needed, OpenTelemetry is used to expose custom application metrics in a Prometheus-compatible format. The collected metrics cover several categories: CPU and memory usage, network traffic, storage activity, system status (including the number of available nodes), as well as additional metrics on power and energy consumption, GPU utilisation (via the NVIDIA GPU operator and dcgm-exporter) and NVIDIA Jetson devices via a custom exporter. These metrics are visualised using Grafana dashboards, which provide both generic Kubernetes views and customised panels for EXTRACT-specific needs.

To support monitoring across multiple clusters, the architecture includes a federated monitoring layer. An additional Prometheus instance is deployed in the cloud cluster, which scrapes the Prometheus servers running in the individual clusters and aggregates their metrics into a single logical view of the compute continuum. On top of this federated storage, WP3 has implemented a REST API using FastAPI and the Prometheus Python client. This API offers a simplified interface for querying metrics over configurable time ranges and resolutions, and is the main entry point for the orchestrator and other EXTRACT components that need monitoring data without dealing with PromQL directly.

In the final integration, the monitoring system and the orchestrator are coupled through a dedicated metrics consumer in the COMPSs runtime and a lightweight monitoring tool deployed alongside the COMPSs master in each workflow Pod. The orchestrator uses the monitoring API to retrieve both infrastructure-level metrics (e.g. node load, available memory, energy usage) and application-level metrics (e.g. task execution times), and incorporates this information into its scheduling and scaling decisions. At the same time, the monitoring architecture is designed to respect security and privacy constraints by limiting the type and granularity of information exposed, and to keep its own performance impact low, in line with the KPI2.2 target of less than 5% overhead.

2.2. COMPSs Orchestrator

The EXTRACT orchestration stack is organized in two layers: an application-level orchestrator and an infrastructure-level orchestrator. This separation of concerns was first introduced in D3.1 and fully implemented in D3.2 and D3.3.

At the application level, EXTRACT relies on COMPSs as the main programming and orchestration model. COMPSs enables data-mining workflows to be expressed as a set of tasks with explicit data dependencies, forming a Task Dependency Graph (TDG). The COMPSs runtime follows a master/worker paradigm and uses the TDG to schedule tasks to workers based on data locality, resource availability and, in the final release, monitoring information. In this context:

- The COMPSs scheduler acts as the application scheduler, deciding on which worker each task should run.
- The COMPSs master runtime is the application orchestrator, responsible for offloading tasks, tracking their execution, and managing dependencies.

At the infrastructure level, the platform uses Kubernetes to manage containerised workloads across the compute continuum. Kubernetes is responsible for deploying the COMPSs master and worker components as Pods, assigning them to nodes according to available resources, and maintaining their lifecycle and connectivity.

In this context:

- The Kubernetes scheduler plays the role of infrastructure scheduler, deciding where Pods are placed.
- The wider Kubernetes control plane acts as the infrastructure orchestrator, handling replication, restarts, and basic service discovery.

The design adopted in WP3 ensures that these two scheduling layers cooperate rather than conflict: COMPSs makes task-level decisions within a pool of workers that Kubernetes maintains and scales.

To enable dynamic resource management during workflow execution, the platform includes a dedicated Monitoring Stack that allows the COMPSs runtime to retrieve scaling decisions and adapt the execution environment accordingly.

The monitoring stack acts as a control layer between the COMPSs runtime and the underlying Kubernetes infrastructure. It collects monitoring information, evaluates scaling policies, and exposes scaling decisions through a lightweight REST interface. The runtime remains responsible for executing the requested scaling operations through its resource management mechanisms. The following figure illustrates the interaction between the COMPSs runtime, the monitoring stack, and the cluster infrastructure.

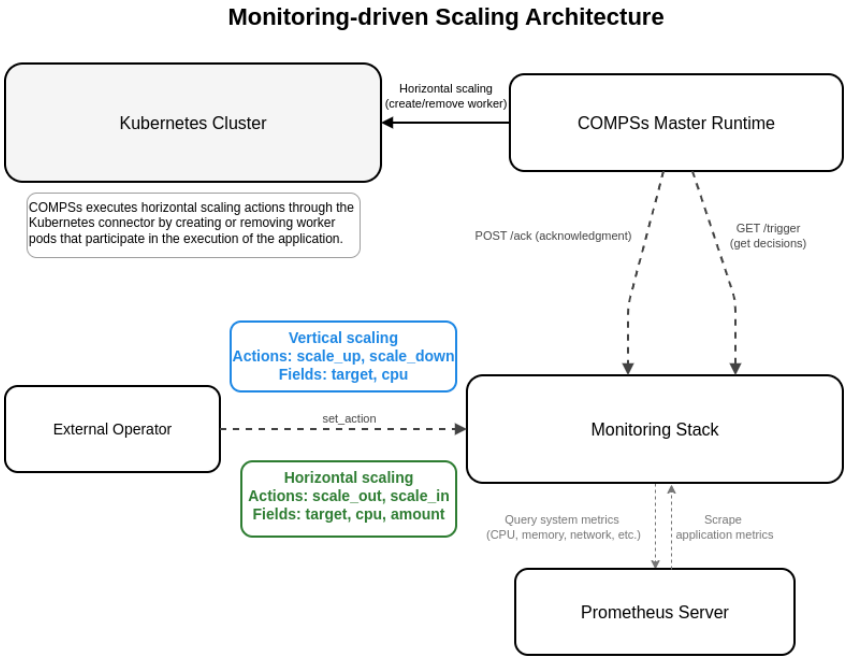

Figure 1: Monitoring-driven Scaling Architecture

The interaction follows a request–acknowledgment mechanism:

- The COMPSs Master Runtime periodically polls the monitoring service via the *trigger* endpoint to obtain scaling decisions.
- The monitoring stack returns a decision payload describing the scaling action to apply.
- After executing the requested operation, the runtime calls the *ack* endpoint to confirm the action was successfully applied.

This mechanism ensures scaling decisions are executed exactly once and prevents repeated execution of the same action.

2.2.1. Monitoring infrastructure interaction

The monitoring stack relies on a Prometheus-based monitoring infrastructure to obtain runtime and infrastructure metrics. Prometheus periodically scrapes application-level metrics exposed by monitoring components and runtime services. In parallel, the monitoring stack queries the Prometheus API to obtain infrastructure-level metrics such as CPU utilisation,memory consumption, and network activity.

These metrics provide the information required to evaluate scaling conditions and determine whether additional resources are required, or whether existing resources can be reduced.

External scaling operator

Scaling decisions may optionally be influenced by an external operator. This component represents an external control entity capable of requesting scaling actions through the monitoring stack.

Depending on the deployment scenario, the external operator may take different forms:

- In some cases, the application itself may act as the operator, requesting additional resources when performance objectives (e.g., execution time constraints or throughput targets) are not met.
- In other scenarios, the operator may correspond to an autonomous control component (e.g., a dedicated containerised controller) that monitors system behaviour and optimises overall resource efficiency across the cluster.

This design enables flexible integration of different policy engines, while keeping the COMPSs runtime responsible for executing the final resource management actions.

Vertical scaling operations

Vertical scaling adjusts the computational capacity of existing workers by modifying their CPU allocation. This allows the runtime to increase or reduce the resources assigned to running workers without changing the number of workers participating in the computation.

Two vertical scaling actions are supported:

- *scale_up*: increases the CPU capacity of a worker
- *scale_down*: reduces the CPU capacity assigned to a worker

Vertical scaling actions are defined using the following parameters:

- *target*: identifies the worker whose CPU resources must be modified (or all workers)

- *cpu*: specifies the CPU resources that should be added or removed from the worker

Horizontal scaling operations

Horizontal scaling adjusts the number of active workers participating in the computation by creating or removing worker Pods in the Kubernetes cluster.

Two horizontal scaling actions are supported:

- *scale_out*: launches additional worker Pods in the Kubernetes cluster
- *scale_in*: removes non-critical worker Pods while preserving the minimum workers required for system operation

Horizontal scaling actions use the following parameters:

- *target*: optionally specifies the node or machine where the worker should be deployed
- *cpu*: defines the CPU configuration assigned to the worker Pods
- amount: indicates the number of worker Pods to be created or removed

The COMPSs runtime executes these operations through its Kubernetes connector, which manages the creation and removal of worker deployments in the cluster.

Runtime interaction with the monitoring layer

The interaction between the runtime and the monitoring stack follows a monitoring-driven control loop: the monitoring infrastructure evaluates metrics and determines whether scaling actions are required; the COMPSs runtime retrieves these decisions, executes the corresponding resource adjustments, and acknowledges their completion.

This architecture enforces a clear separation between decision-making logic and resource execution. As a result, scaling policies can evolve independently of the runtime while still allowing COMPSs to dynamically adapt the execution environment to workload variations and infrastructure conditions.

2.3. Event-driven Dynamic Workflow Orchestration as a PyCOMPSs Extension

The workflow execution layer is organised around a dedicated orchestration component that separates workflow description from task supervision and runtime control. This layer is implemented through the WorkflowManager, the Task_Wrapper, the MQTT communication layer, and a set of example workflows that combine sequential dependencies, parallel branches, and iterative retry loops.

Instead of invoking task logic directly from the master process, the framework submits PyCOMPSs tasks using symbolic metadata (module and function names). This design enables the orchestration layer to dynamically resolve execution callables, associate each invocation with a unique task instance identifier, and maintain explicit bookkeeping for task futures and logical task names.

2.3.1. Workflow manager responsibilities

The WorkflowManager acts as the central coordination entity for workflow submission and completion tracking. For every logical task (defined as the execution task plus its implicit evaluation function), it:

1. Extracts the module and function names of both the execution function and its evaluation function.
2. Generates a unique *task_instance_id*.
3. Stores the corresponding future.
4. Emits a *task_submitted* event containing the task metadata and call index.

The manager also supports evaluation-specific argument injection, allowing a common submission interface to be adapted to heterogeneous control policies.

Completion handling is performed through the *wait_for_tasks* method, which consumes task-finished notifications from MQTT and materialises the associated PyCOMPSs futures using *compss_wait_on*. If any task evaluation returns False, the manager publishes a global cancellation flag, emits a *workflow_repeat_request* event, and aborts the current iteration early after enforcing a COMPSs barrier.

Processed futures are then removed from internal tables to ensure that subsequent sequential phases or retry iterations do not block on stale task handles.

2.3.2. Task isolation and cancellation-aware execution

The Task_Wrapper is declared as a PyCOMPSs task that returns a boolean status. However, the user function and its evaluation are executed inside a separate child process created via Python multiprocessing. This additional level of indirection:

1. Isolates user code from the parent task process.
2. Enables external termination through operating-system signals.
3. Provides a controlled mechanism for returning both the function result and the evaluation outcome through a multiprocessing queue.

Before launching the child process, the wrapper resolves the target execution and evaluation callables dynamically from their module and symbol names. During execution, the parent process periodically polls the MQTT cancellation flag and, when cancellation is requested, sends *SIGTERM* either to the child process group or to the process itself. The wrapper can also trigger a Lithops cleanup, terminating any Lithops jobs that may have been started from within the task.

If the child process terminates normally, the wrapper retrieves the queue payload and distinguishes among:

- Successful execution
- Explicit exceptions
- Silent termination without a returned result.

In all cases, it publishes a per-instance *DONE* notification and emits a structured *task_finished* event containing the task identity, cancellation status, a stringified function result, and the evaluation result.

The wrapper also includes a defensive instrumentation-suppression routine for pyextrae, motivated by process-local tracing issues that can occur after forking within the PyCOMPSs worker inner context.

2.3.3. MQTT-based control plane

A lightweight MQTT control plane is used to coordinate master-side orchestration, worker-side completion signals, manual decisions, and telemetry publication.

Dedicated topics are defined for workflow cancellation, task completion, manual evaluation, workflow-wide run identifier dissemination, and structured events, in order to avoid cross-talk. The master listens asynchronously for *DONE* notifications from the workers and pushes received task instance identifiers into a local queue. This decouples completion detection from directly blocking on every future and allows the orchestrator to react to the first evaluation failure as soon as it is reported.

The resulting communication model is intentionally simple, but it provides a clear separation between execution data, control-state propagation, and observable workflow telemetry.

3. Validation Methodology

This section presents the methodology used to validate the main runtime mechanisms implemented in the EXTRACT orchestration stack. The validation focuses on demonstrating functional correctness and expected operational behavior under controlled and repeatable conditions, rather than providing an end-to-end benchmark of full application workloads.

3.1. Monitoring-driven Scaling Architecture

This section validates the monitoring-driven scaling mechanisms integrated into the EXTRACT orchestration stack. The objective is to assess the correctness and behavior of horizontal and vertical scaling operations under controlled conditions.

The scaling architecture supports:

- Horizontal scaling: scale-out and scale-in of COMPSs worker Pods.
- Vertical scaling: dynamic adjustment of CPU resources assigned to workers.

Scaling operations are triggered based on monitoring data collected from the Prometheus infrastructure and processed by the monitoring stack.

Validation is performed using a synthetic workload (a matrix multiplication application) designed to exhibit clear computational phases and sustained resource usage. This controlled setup allows scaling effects to be observed in isolation, independently of the complexity of full workflows.

Figure 2 presents a Paraver trace capturing multiple scaling scenarios within a single execution. The trace shows:

- A scale-out phase, where an additional COMPSs worker is deployed with 2 CPUs;
- A scale-up phase, where the CPU allocation of the additional worker is increased from 2 to 4 CPUs (+2 CPUs);
- A scale-down phase, where the CPU allocation of the additional worker is reduced from 4 back to 2 CPUs (−2 CPUs); and
- A scale-in phase, where the additional worker is removed, restoring the initial configuration.

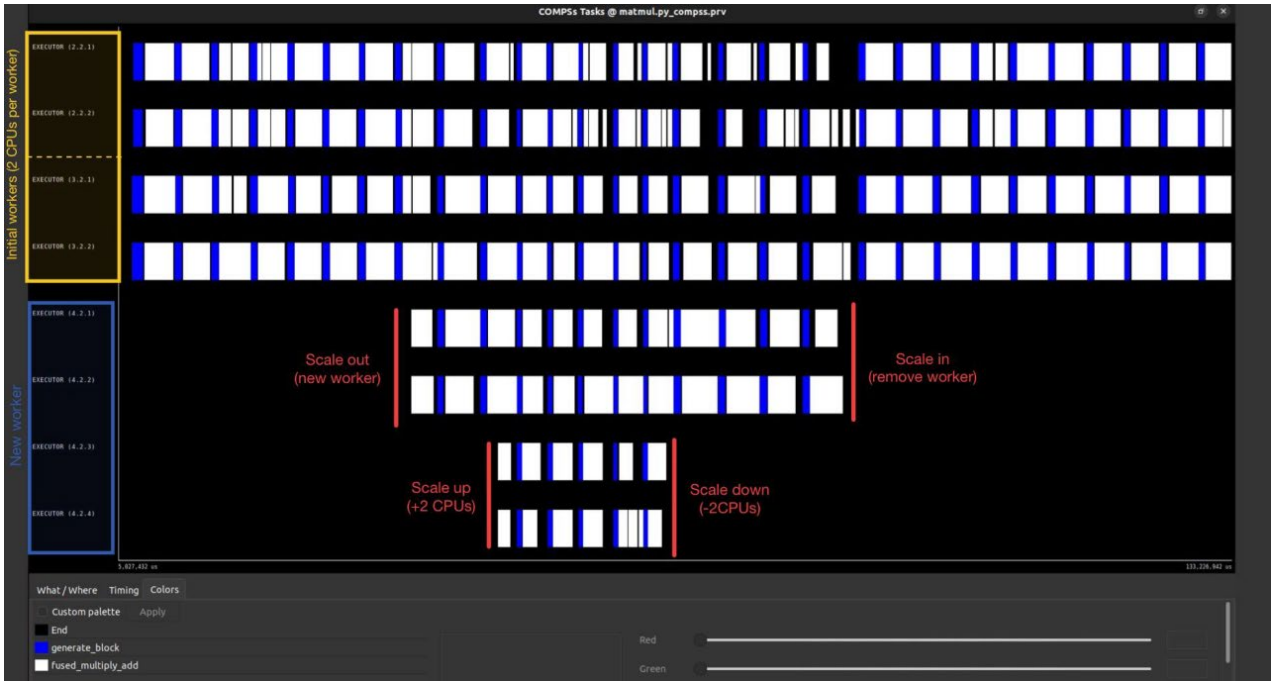

Figure 2: Paraver trace of a matrix multiplication workload illustrating monitoring-driven horizontal and vertical scaling.

The trace reflects how the COMPSs runtime adapts to scaling decisions provided by the monitoring layer. Horizontal scaling increases the number of concurrent execution lanes, whereas vertical scaling modifies the level of intra-node parallelism available within a worker. Resource removal produces a visible reduction in concurrent task execution.

Although these scaling mechanisms are validated independently, they are not actively exploited in the TASKA-C workflow presented in Section 4. This separation isolates validation of the scaling architecture from the behaviour of specific workflows.

3.2. Event-driven Dynamic Workflow Orchestration as a PyCOMPSs Extension

Manual evaluation and iterative execution

The implemented framework supports human-in-the-loop decision points through the manual_evaluation function. When this evaluation mode is selected, the task publishes a *manual_eval_requested* event that includes the MQTT topic on which an external user or client must send either *CONTINUE* or *ABORT*.

This human-in-the-loop mechanism enforces an automatic blocking behavior: any running or already scheduled tasks are allowed to proceed up to the logical decision point (e.g., the boundary of a user-defined retry loop), but no further tasks are executed until the human notification is received. If all evaluations succeed, the loop terminates and the workflow proceeds to the next stage. Conversely, a failed automatic or manual evaluation causes the current parallel region to be cancelled and repeated.

The supplied sequential and nested examples demonstrate two complementary orchestration patterns:

- The sequential workflow alternates submission and waiting, making dependency progression explicit.
- The nested workflow submits independent branches after a common predecessor and then waits for their completion as a group.

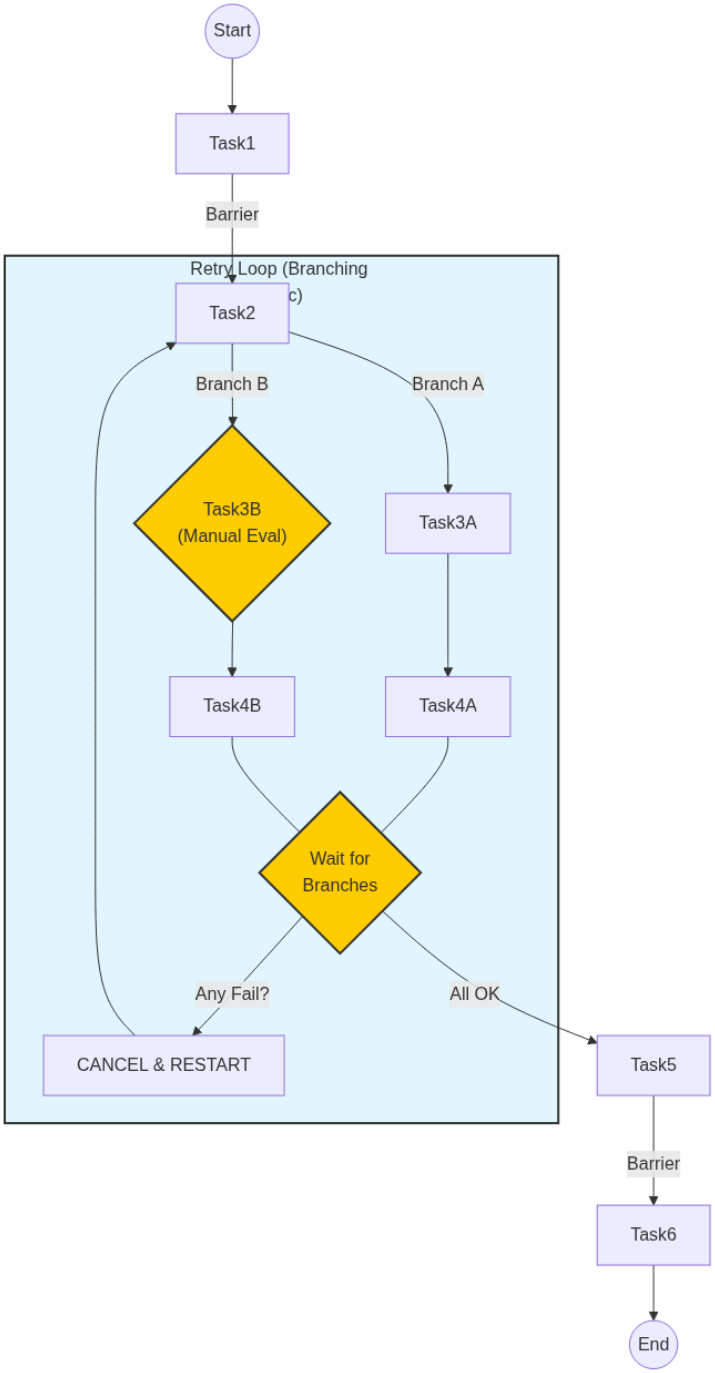

Figure 3: Graphic representation of the dynamic workflow orchestration

Together, these examples show that the orchestration layer supports both strictly ordered pipelines and partially parallel retry regions without changes to the underlying control mechanisms.

4. TASKA Use Case – End-to-End validation

4.1. Architecture Overview

The TASKA workflow in EXTRACT is executed through the integration of COMPSs as the application-level orchestrator and Lithops as the serverless execution backend. This integration enables the deployment of data-intensive workflows across the computing environment while preserving a high-level programming and orchestration model.

At the application level, COMPSs represents the workflow as a Task Dependency Graph (TDG), where tasks are scheduled according to data dependencies and resource availability. At the execution level, Lithops provides a unified abstraction for Function-as-a-Service (FaaS) and batch execution, enabling COMPSs tasks to be offloaded to distributed execution environments.

This execution model is supported by Kubernetes as the infrastructure layer, which manages the lifecycle of COMPSs master and worker components, and by the monitoring stack, which continuously collects runtime and infrastructure metrics. Together, these components form a layered orchestration architecture in which task scheduling, execution, and monitoring are integrated

4.2. Execution Model

The integration between COMPSs and Lithops enables a distributed execution model in which workflow tasks are transparently executed on remote resources while maintaining a consistent programming abstraction.

When a task is triggered by the COMPSs runtime, it can be offloaded to Lithops. Lithops is then responsible for provisioning execution environments, dispatching tasks, and managing data transfers between storage and compute resources. This approach decouples computation from the underlying infrastructure, allowing the same workflow to run on different backends without requiring workflow modifications.

From the developer perspective, this complexity is hidden behind the COMPSs programming model: workflows can be expressed as sequential Python code while still benefiting from distributed execution.

Internally, the workflow execution structure is defined by the TDG managed by COMPSs. The graph explicitly represents task dependencies, synchronisation points, and control-flow constructs such as barriers and evaluation tasks.

Figure 4 illustrates the TDG of the TASKA-C workflow, including the control and evaluation tasks described earlier. Synchronisation points and barriers ensure correct execution ordering, while evaluation tasks enable conditional progression and potential workflow cancellation.

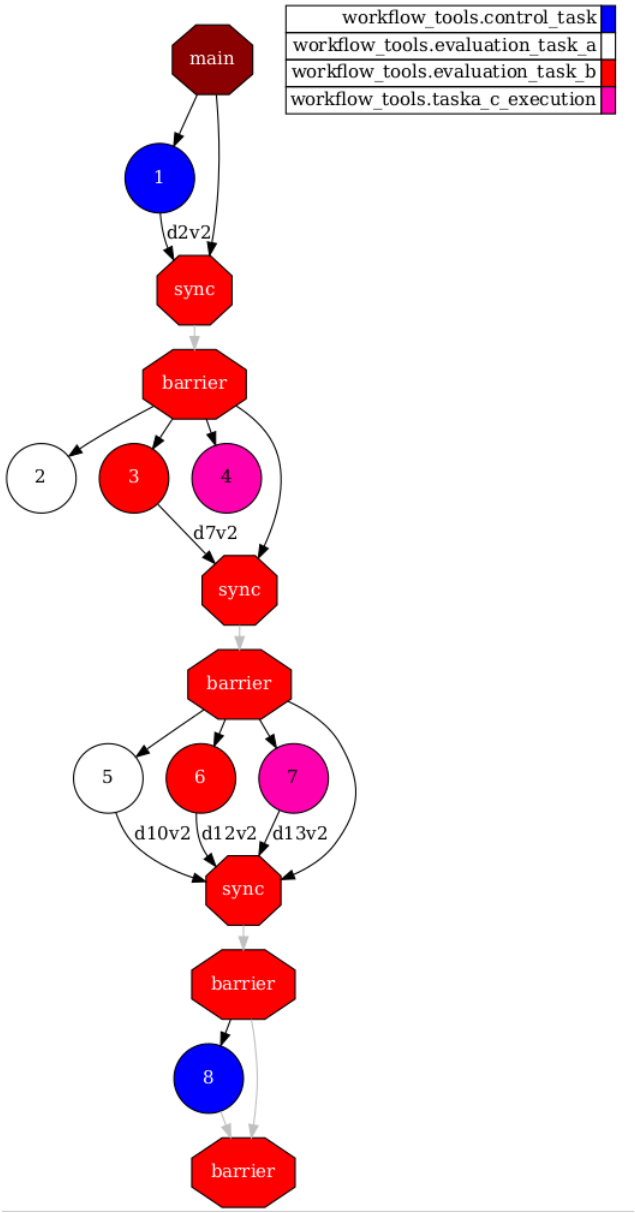

Figure 4: COMPSs task graph of the TASKA-C workflow with evaluation and cancellation behaviour.

4.3. Integration with Orchestration

The COMPSs–Lithops execution model is integrated into the EXTRACT data-driven orchestration framework through a monitoring-driven control loop, in which runtime and infrastructure metrics are collected and made available to the orchestration layer.

The monitoring infrastructure provides information about resource utilisation, task execution behaviour, and overall system status across the execution environment. This information supports informed decision-making and enables adaptive execution strategies.

Although the EXTRACT architecture supports monitoring-driven scaling mechanisms, these capabilities are not actively exploited in the TASKA-C workflow due to its execution characteristics. The scaling architecture is instead validated independently, as described in Section 3.1.

Nevertheless, the integration between COMPSs, Lithops, and the monitoring infrastructure establishes the foundation required to enable such adaptive behaviours. The design allows monitoring-driven mechanisms to be incorporated with minimal changes should future workflow requirements justify their use. A detailed description of the monitoring-driven scaling architecture is provided in Section 3.1.

4.4. Validation in the TASKA C Use Case

The TASKA-C workflow is executed end-to-end on a Kubernetes-based infrastructure, where the required components are deployed using Helm charts. This includes the COMPSs master and worker components, as well as the supporting services required for workflow execution. This setup provides a realistic execution environment in which the workflow can be orchestrated and monitored, enabling analysis of runtime behaviour under representative conditions.

The workflow consists of multiple stages with heterogeneous computational characteristics, including I/O-intensive, memory-bound, and compute-intensive tasks. This heterogeneity makes TASKA-C suitable for analysing execution behaviour under different workload conditions.

To analyse runtime behaviour, execution traces were collected during workflow execution. Figure 5 presents a Paraver trace of the TASKA-C workflow, showing the temporal distribution of task execution across workers and the interaction between execution and evaluation phases. The trace highlights how tasks are scheduled, executed, and synchronised, and how evaluation tasks influence workflow progression.

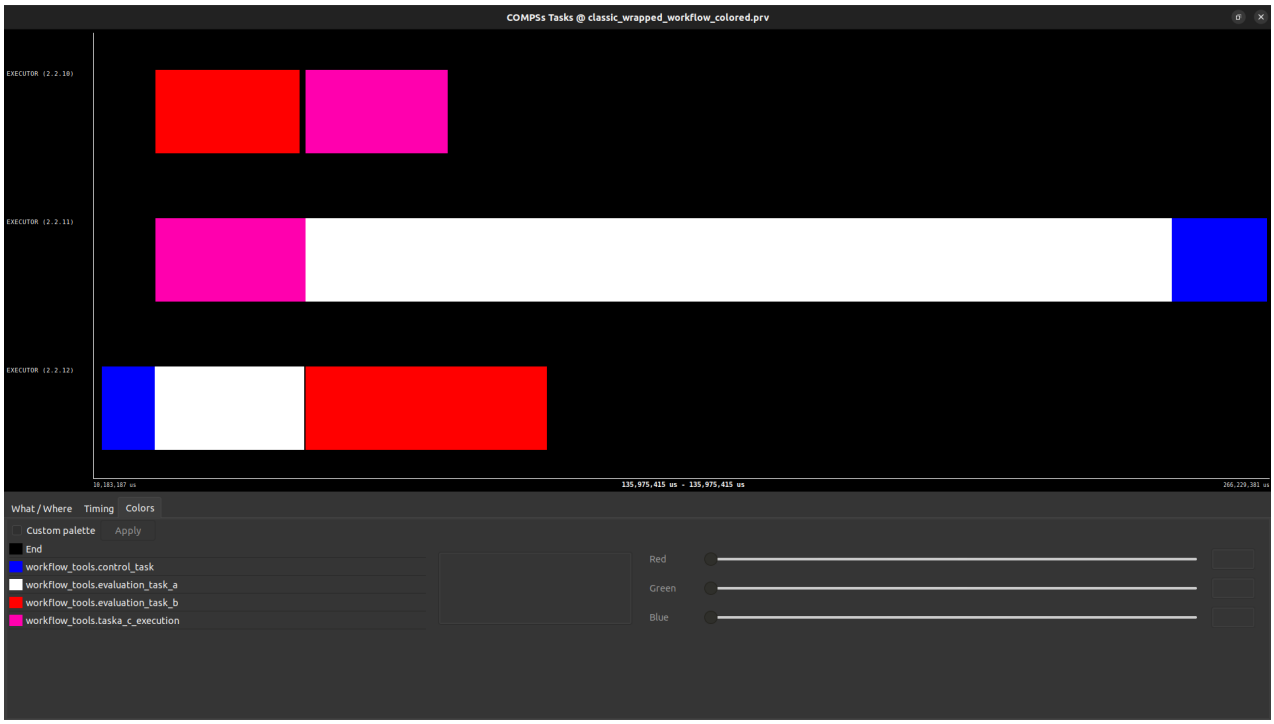

Figure 5: Paraver trace of the TASKA-C workflow showing task execution and cancellation behaviour.

The trace shows that task execution is organised into phases separated by synchronisation points, reflecting the structure defined in the Task Dependency Graph (TDG). Parallel execution is observed in stages where independent tasks can be processed concurrently, whereas other stages exhibit sequential behaviour due to data dependencies or control constraints.

Although the deployment uses a single COMPSs worker Pod, parallelism is achieved through the utilisation of multiple CPU cores within the node. The COMPSs runtime exploits intra-node parallelism by scheduling independent tasks across the available computational resources. This behaviour is reflected in the Paraver trace, where concurrent task execution can be observed despite the presence of a single worker.

In addition to execution traces, monitoring data was collected to analyse resource utilisation across the execution environment. Figure 6 presents CPU usage collected from the monitoring infrastructure (Prometheus) during execution of the TASKA-C workflow. The monitoring data captures the resource utilisation of both the COMPSs master and worker Pods over time, showing how CPU consumption evolves across different workflow phases. Differences in CPU usage patterns between master and worker components are visible and reflect their distinct roles in orchestration and execution.

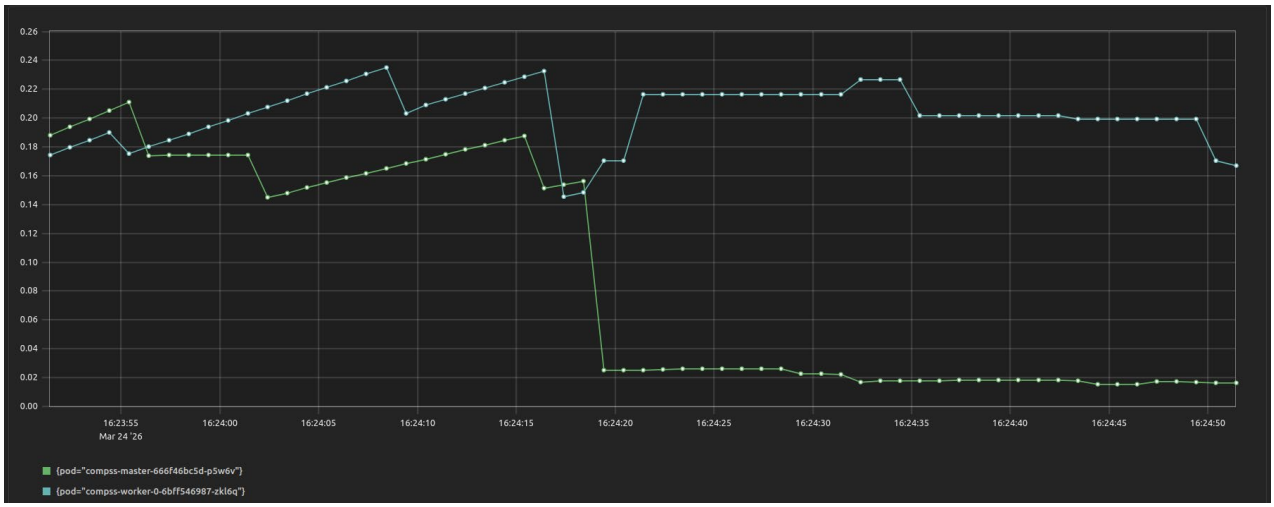

Figure 6: CPU usage of a COMPSs worker pod during the execution of the TASKA-C workflow, reflecting intra-node parallelism across multiple CPU cores.

The observed CPU patterns are consistent with the execution behaviour shown in the Paraver trace. Periods of increased utilisation correspond to phases where multiple tasks execute concurrently across CPU cores, while lower utilisation is observed during synchronisation points and control phases.

This correlation between execution traces and monitoring data provides a more complete view of workflow behaviour by combining task-level execution dynamics with infrastructure-level resource usage.

4.5. Impact on Orchestration Efficiency

The integration of COMPSs with Lithops within the EXTRACT orchestration framework enables efficient execution of data-intensive workflows by aligning the execution model with the structure of the workflow.

The monitoring-driven approach provides visibility into runtime behaviour and supports identification of execution patterns such as parallel phases, synchronisation points, and control-driven transitions.

Although dynamic scaling is not applied in this workflow, the observed execution characteristics highlight the potential for adaptive resource management strategies. The clear separation between execution phases, together with variability in task behaviour, suggests that different stages could benefit from differentiated resource configurations.

Overall, the validation demonstrates that the COMPSs–Lithops integration provides a solid foundation for data-driven orchestration, enabling structured, observable, and potentially adaptable workflow execution.

5. PER Use Case – End-to-End Validation

5.1. Architecture Overview

The PER use case validates end-to-end execution of a continuum-oriented workflow implemented with PyCOMPSs/COMPSs and deployed across a Kubernetes multiclusterenvironment. The workflow models a personalised evacuation system over a city-scale street network under emergency conditions, combining (i) high-fidelity graph representations derived from OpenStreetMap (OSM), (ii) deterministic city simulation, and (iii) reinforcement-learning (RL) control policies into a single task-based pipeline.

The design naturally maps onto the compute continuum: compute-intensive trainingis executed on HPC resources(GPU-enabled nodes), while latency-sensitive inferenceis served on edge nodes. PyCOMPSs provides the task-based programming abstraction used to express this distributed application as annotated Python functions, from which COMPSs builds a Task Dependency Graph (TDG)and orchestrates execution based on data dependencies and resource constraints.

Overall, the architecture integrates:

- COMPSs (application-level orchestration): task dependency management, scheduling, and execution supervision.
- Kubernetes multiclusters (infrastructure layer): isolated clusters representing HPC and edge sites, each running COMPSs components as Deployments and Services.
- Multicluster connectivity (service mirroring): cross-cluster service discovery and communication enabling remote workers to register to, and interact with, the master.

5.2. Execution model

The PER workflow is structured as an end-to-end RL-based pipeline composed of four stages:

1. **Map generation**. This stage produces an OSM-based layered graph that defines the training topology. The map is serialised once and then reused read-only by subsequent tasks.
2. **Parallel training.** The objective is to learn a routing policy that minimises total evacuation time while avoiding congestion and danger nodes. Training uses a Branching Dueling Q-Network (BDQ) architecture, suitable for high-dimensional discrete action spaces. To address scalability limitations of monolithic training on large topologies, training is distributed across (N) workers initialised from a common model and consuming the same serialised topology. In each training round, each worker executes approximately (T_e/N) episodes on a GPU-enabled node. Since exploration is stochastic, workers collect complementary experience and contribute to faster convergence.

3. **Model aggregation.** Locally trained models are synchronised using a model-averaging approach inspired by Federated Averaging, producing an aggregated global model. The aggregated model is evaluated; if target performance is not reached, the model is redistributed and training iterates for another round.
4. **Inference serving.** Once the aggregated model satisfies the target criterion, it is packaged behind a lightweight service that can be queried externally. This stage is intended to run close to users (edge tier) to reduce latency.

From an orchestration standpoint, PyCOMPSs translates these stages into a TDG where:

- The map generation task is a single predecessor.
- The training stage forms a parallel region (e.g., (N=3) parallel training tasks in the evaluated configuration).
- Aggregation and convergence evaluation introduce synchronisation and control-flow.
- Inference tasks are scheduled onto edge resources according to explicit constraints.

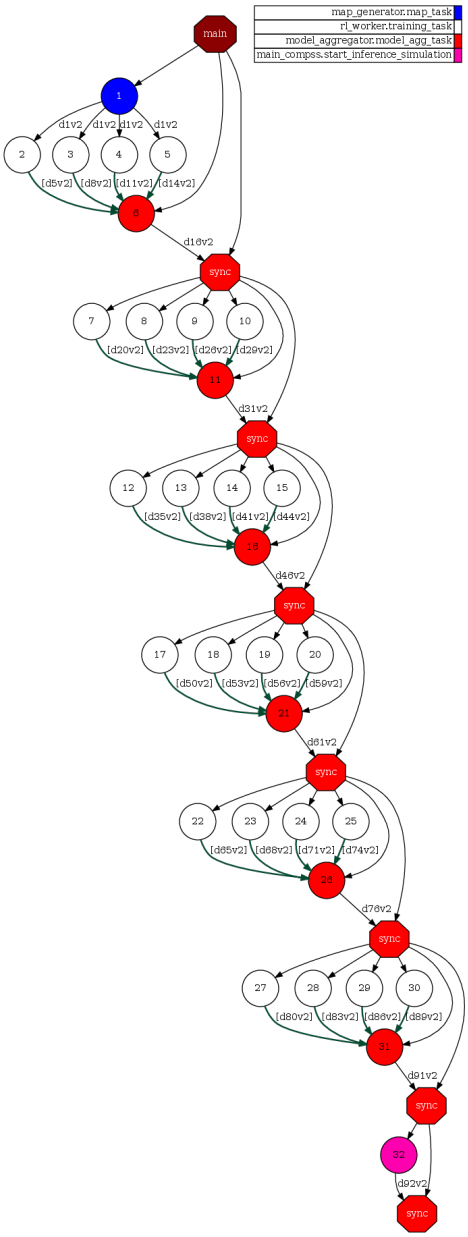

Figure 7: COMPSs task graph of the PER workflow.

5.3. Integration with Orchestration

The workflow is deployed using a Kubernetes multicluster abstraction via the kubecompss CLI, allowing the application deployment to be described declaratively and executed in a single step. The user specifies the Kubernetes contexts involved (HPC and edge clusters) and the COMPSs components to be deployed per cluster; kubecompss generates and applies the corresponding Kubernetes resources across clusters.

In the evaluated setup:

- One COMPSs master and one worker are deployed in the HPC cluster.
- An additional worker is deployed in the edge cluster.

Each component runs as a Kubernetes Deployment and is exposed through a Service for stable networking, as required by the COMPSs runtime.

Cross-cluster integration is achieved through service mirroring (e.g., Linkerd Service Mirroring): the edge worker service is exported and mirrored into the HPC cluster, and the master service is mirrored into the edge cluster. This enables:

- The master to discover and register both local and remote workers transparently.
- Remote workers to communicate with the master without application changes.

This integration provides a unified orchestration domain where COMPSs performs application-level scheduling over a pool of workers spanning multiple clusters, thus operationalising the compute-continuum abstraction.

5.4. Validation in the PER Use Case

We have evaluated the execution times of the PER workflow to quantify the performance impact of executing it on the compute continuum using COMPSs and Kubernetes multiclusters. We compare three deployment strategies:

- Sequential (A). The workflow is executed end-to-end on a single site without task-level parallelism.
- Kubernetes without a programming model. We use this method as a *theoretical lower bound* rather than an implemented system. It represents an idealized manual Kubernetes deployment in which the developer explicitly parallelizes the parallel training stage across workers, with negligible orchestration overhead.
- Programming model on Kubernetes multiclusters (C). The workflow is executed with COMPSs across the multicluster setup using Service Mirroring for cross-cluster connectivity. The training stage is executed with $N=3$ parallel training tasks.

To make the three methods comparable, we derive the lower bound (B) from the sequential baseline (A) using the same degree of parallelism as the COMPSs run. In particular, for the parallel training stage, the sequential configuration runs a total of 1000 episodes in a single process, whereas the COMPSs configuration distributes the same per-round budget across three workers, executing 333 episodes per worker.

The theoretical lower bound assumes ideal scaling for training and is therefore computed as B = A/3. For stages that are not parallelized in our workflow (map generation, model aggregation, and inference serving) we assume no theoretical speedup and thus set B = A. This makes B an optimistic baseline: it ignores dependency tracking, scheduling, synchronization, container startup, data transfers, and multicluster networking effects.

As a consequence, the measured COMPSs multicluster execution is expected to satisfy C > B. The difference C - B captures the **overhead** paid to obtain programmability and compute-continuum abstraction. Conversely, the difference A - C quantifies the **practical benefit** over sequential execution due to runtime-managed parallelism in training.

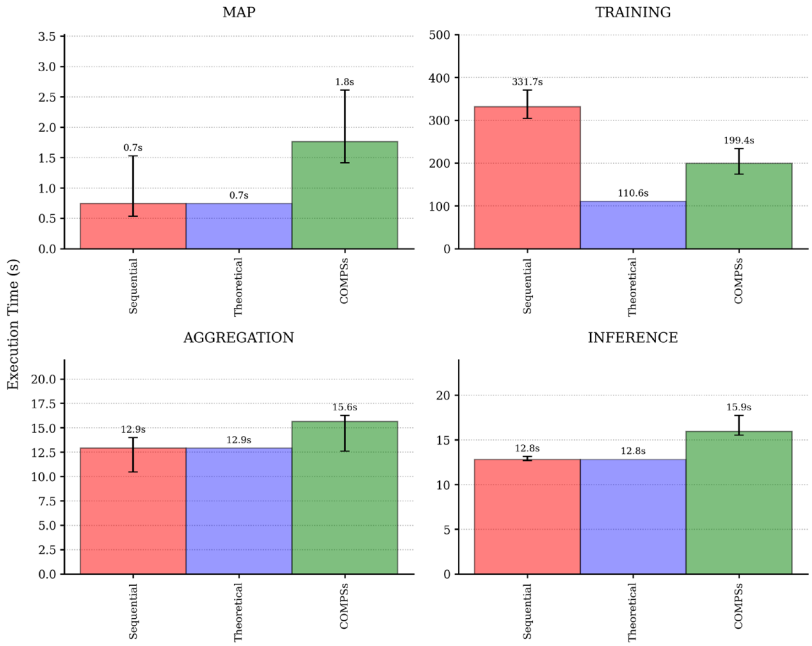

Figure 8: Bar charts comparison between tasks and deployment strategies.

Figure 8 reports per-stage execution times for the three deployment strategies. For map generation, aggregation, and inference, the comparison primarily exposes task-based runtime overhead (since B = A), whereas for training it additionally quantifies how much of the ideal 3 x parallel speedup is realized in practice.

6. Cross-Use-Case Assessment of Objective O2

This section provides a consolidated assessment of Objective O2across the validated use cases and platform components described in this deliverable. O2 targets data-driven orchestration across the compute continuum, with two complementary dimensions: (i) selecting and managing resources to satisfy data and execution requirements efficiently (O2.1 / KPI2.1) and (ii) enabling monitoring to inform such decisions while preserving confidentiality and limiting runtime overhead (O2.2 / KPI2.2). The assessment leverages the evidence produced in the scaling validation (Section 3), the event-driven orchestration extension (Section 3.2), and the end-to-end workflow validations (TASKA in Section 4 and PER in Section 5).

6.1. Global assessment of KPI2.1

KPI2.1. Efficient and secure processing of extreme data, by selecting the most appropriate computing resources that guarantee the fulfilment of data characteristics (including security), while minimizing power consumption.

Across the EXTRACT project, KPI2.1 has been addressed through the combination of:

1. A layered orchestration architecture (COMPSs at application level and Kubernetes at infrastructure level)
2. Explicit placement and execution abstractions (COMPSs scheduling, Kubernetes worker pools, and—where applicable—continuum placement mechanisms).
3. Dynamic resource management primitives (horizontal and vertical scaling). Together, these elements enable the platform to select and adjust computing resources according to workload demands and operational constraints.

Evidence from the monitoring-driven scaling validates the correctness of both, horizontal (scale-out/scale-in of workers) and vertical (scale-up/scale-down CPU allocation per worker), scaling operations under controlled conditions. The Paraver trace confirms that:

- Scale-out increases the number of concurrent execution lanes (more parallel task execution capacity);
- Scale-up increases intra-worker parallelism by raising CPU allocation; and
- Scale-down/scale-in reduce available resources, visibly lowering concurrent execution.

This validation demonstrates that the EXTRACT orchestration layer can adapt allocated resources during runtimein a controlled, observable, and repeatable manner. Although this experiment does not directly quantify energy savings, the availability of reversible scaling actions is a necessary prerequisite for power-aware policies (e.g., reducing resources during low-load phases or avoiding overprovisioning).

Moreover, in the TASKA-C workflow execution, we show an end-to-end execution on Kubernetes with COMPSs as orchestrator and Lithops as data parallelization runtime. While dynamic scaling is not appliedin the TASKA-C run, the collected traces and monitoring data show clear phase separation, alternating parallel computation and synchronisation/control phases. This observation is important for KPI2.1 because it highlights that the workflow exhibits stage-dependent behaviour (I/O-intensive, memory-bound, compute-intensive), which is a prerequisite to justify differentiated resource configurations per stage (e.g., allocating more CPU during parallel compute phases and reducing allocations around barriers and evaluation points).

In other words, TASKA-C validates that the platform can provide structured and observable execution, which is a key enabler for selecting the most appropriate resources based on the phase and the implied data/compute characteristics, even if it does not yet apply an energy-optimising policy to this workflow.

The PER workflow explicitly demonstrates continuum-style placement: compute-heavy RL training is executed on HPC (GPU-enabled) resources, while latency-sensitive inference servingis placed closer to users at the edge. This supports KPI2.1 in two ways:

- It shows that the orchestration model can map heterogeneous stages to heterogeneous resources (performance-driven placement); and
- It provides a concrete basis for energy/performance trade-offs, since training and inference have different latency and throughput objectives and can be provisioned differently.

The per-stage execution time comparison presented for PER further quantifies the practical benefits and overheads of runtime-managed parallelism. The measured reduction in training time from the sequential baseline confirms that distributing the training phase across multiple workers improves time-to-solution, while the gap to an optimistic theoretical lower bound reflects orchestration and coordination costs that must be considered when optimising for energy efficiency.

Assessment against the KPI2.1 measurement approach (Phase 2 vs Phase 3). The deliverable provides the key architectural and validation building blocks required by KPI2.1:

- Mechanisms to vary resources (horizontal/vertical scaling validated in isolation).
- Observability of workflow phases and runtime behaviour (traces and metrics in TASKA-C).
- Continuum placement rationale (PER: HPC for training, edge for inference).

6.2. Global assessment of KPI2.2

KPI2.2. Distributed monitoring architecture capable of continuously collecting the required information from the EXTRACT platform and pass it to the data-driven orchestrator, without revealing information that could inadvertently expose internals of the workflow under monitoring and with minimum impact on performance.

KPI2.2 is addressed by the monitoring architecture described and exercised in Sections 2 and 3, and by the way monitoring outputs are used as part of orchestration validation in Sections 4 and 5.

The EXTRACT implements a Prometheus-based monitoring infrastructure in which:

- Prometheus scrapes application-level metrics and monitoring component endpoints.
- The monitoring stack queries Prometheus to retrieve infrastructure-level metrics (CPU, memory, network).
- The monitoring stack evaluates policies and exposes scaling decisions via a lightweight REST interface.
- The COMPSs master periodically polls decisions (trigger) and acknowledges execution (ack), ensuring exactly-once application of scaling actions.

This validates that monitoring is not treated as an external afterthought, but as an integrated control-plane component that can drive orchestration decisions while keeping execution responsibility within the runtime.

From an architectural standpoint, KPI2.2's confidentiality aspect is supported by the design choice that the monitoring layer operates primarily on aggregated runtime and infrastructure metrics (CPU, memory, etc.) and returns only coarse-grained scaling decisions to the runtime. In addition, the event-driven orchestration mechanisms described for the PyCOMPSs extension rely on well-scoped control topics and structured events, providing separation between control-state propagation and execution data.

Evaluation demonstrates operational monitoring integrated with orchestration and uses it in validation, but it does not report a dedicated measurement campaign that quantifies monitoring overhead and verifies it remains below 5% for the validated workflows. Within the text's scope, the claim is supported indirectly by the adoption of standard, low-overhead monitoring patterns (scraping + querying) and by the fact that end-to-end workflows are executed and traced successfully with monitoring enabled. A quantitative confirmation would require explicit A/B experiments with monitoring enabled vs disabled, using comparable runs and reporting deltas for workflow makespan and/or task throughput.

6.3. Main lessons learned

Separation of concerns is essential for predictable orchestration. The two-level orchestration design (COMPSs for task-level scheduling; Kubernetes for pod-level placement and lifecycle) enables cooperation rather than conflict: Kubernetes provides a stable, scalable worker pool, while COMPSs optimises task placement and dependency-aware execution within that pool.

Workflow phase structure creates opportunities for adaptive strategies—even when scaling is not yet applied. TASKA-C shows execution phases, barriers, and evaluation-driven transitions. Even without applying scaling, this observability is valuable: it indicates where adaptive resource allocation could improve efficiency (e.g., allocate more resources during parallel phases and reduce during control/synchronisation phases).

Continuum placement is best justified by clear stage heterogeneity. PER demonstrates a strong continuum case: training is compute/GPU intensive and suits HPC; inference is latency sensitive and suits the edge. This separation provides a natural basis for resource selection policies aligned with performance objectives and, potentially, power optimisation.

Monitoring interfaces have to remain decision-oriented and lightweight. The trigger/ack scaling loop supports clean integration between monitoring-driven decisions and runtime-executed actions. Keeping monitoring outputs at the level of scaling "instructions" helps limit information exposure and simplifies exactly-once execution semantics.

7. Conclusion

This deliverable presented the final design and validation of the EXTRACT orchestration stack, structured around a clear separation between application-level orchestration and infrastructure-level orchestration. At the application layer, COMPSs/PyCOMPSs expresses workflows as task graphs with explicit dependencies and manages task scheduling and execution through a master/worker runtime. At the infrastructure layer, Kubernetes provides lifecycle management, placement, and resiliency for the COMPSs components across the compute continuum. This separation enables the two scheduling layers to cooperate: Kubernetes maintains and scales the worker pool, while COMPSs performs dependency-aware task scheduling within it.

A central outcome of the work is the integration of a monitoring-driven control loop that bridges runtime behaviour and infrastructure management. A Prometheus-based monitoring pipeline feeds a monitoring stack that evaluates policies and exposes scaling decisions via a lightweight REST interface. The COMPSs runtime applies these decisions through a request–acknowledgment mechanism, ensuring scaling actions are executed exactly once. Horizontal and vertical scaling operations were validated under controlled conditions, demonstrating correct resource addition/removal and CPU reconfiguration and their observable effects on execution concurrency.

Beyond infrastructure adaptation, the deliverable introduced and validated an event-driven workflow orchestration extension for PyCOMPSs, enabling dynamic control constructs such as iterative execution, early cancellation, and human-in-the-loop decision points. The design isolates user code execution within child processes to support cancellation-aware behaviour and integrates MQTT-based signalling to decouple completion detection and control-state propagation from direct future blocking.

End-to-end validation was then performed on representative workflows. The TASKA-Cuse case demonstrated the execution of a heterogeneous, multi-stage workflow on Kubernetes, supported by execution traces and monitoring data that jointly characterise task-level behaviour and infrastructure-level resource usage. The PER use case validated continuum-oriented placement and multicluster execution, showing how compute-heavy RL training can exploit HPC resources while inference serving targets edge resources, and quantifying the practical benefits and overheads of task-based parallel execution relative to sequential execution and an idealised lower bound.

Overall, the results demonstrate that EXTRACT provides a solid foundation for structured, observable, and adaptable workflow execution across the compute continuum, combining task-based programming and orchestration with cloud-native deployment and monitoring-driven resource management.

8. Acronyms and Abbreviations

ACK - Acknowledgment

API - Application Programming Interface

BDQ - Branching Dueling Q-Network

CLI - Command-Line Interface

CPU - Central Processing Unit

DDQN - Double Deep Q-Network

DQN - Deep Q-Network

FaaS - Function-as-a-Service

GPU - Graphics Processing Unit

HPC - High-Performance Computing

HTTP - Hypertext Transfer Protocol

I/O - Input/Output

KPI - Key Performance Indicator

k8s - Kubernetes

MQTT - Message Queuing Telemetry Transport

OSM - OpenStreetMap

REST - Representational State Transfer

RL - Reinforcement Learning

SIGTERM - Termination signal (POSIX)

TDG - Task Dependency Graph

WP - Work Package

9. References

[1] PyCOMPSs Documentation, "PyCOMPSs overview," accessed 2026-03-26. (pycompss-poster.readthedocs.io)

[2] Linkerd official documentation. (https://linkerd.io/)

[3] FastAPI official documentation. (https://fastapi.tiangolo.com/)

[4] Grafana Labs Documentation. Prometheus data source. (grafana.com)

[5] Prometheus official documentation. (https://prometheus.io/)

[6] Helm official documentation. (https://helm.sh/)

[7] M. Haklay and P. Weber, "OpenStreetMap: User-Generated Street Maps," IEEE Pervasive Computing, vol. 7, no. 4, pp. 12–18, 2008, DOI:10.1109/MPRV.2008.80