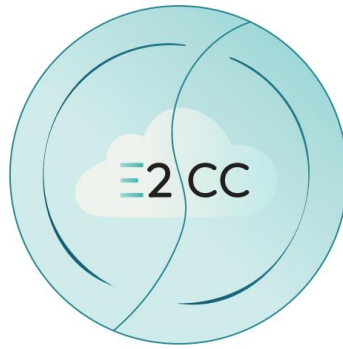




D2.3 Cloud-Edge orchestration solutions considering security, resource management, and energy efficiency, phase 1.

Propriétés du document

Numéro de contrat	DOS0235220/00 – DOS0235222/00
Niveau de dissémination	<Public>
Nature	REPORT
Auteurs	Damien Monguillon, Aline Brouard, Tarek MENOUEUR, Rachid Zarouali, Anne-Cécile Orgerie
Date	26 Juin 2026
Etat	Final



Ce projet a été financé par l'Etat dans le cadre de France 2030.

Historique des changements

Version	Date	Auteur, Organisation	Description des Changements
0.1	11/12/2025	Damien Monguillon, NBS	Initialisation du document
0.1	19/05/2026	Aline Brouard, Armadillo	Traduction en anglais
0.1	11/05/2026	Tarek MENOUEUR	Contribution CGI
0.2	17/06/2026	Damien Monguillon	Schéma d'architecture globale
0.2	22/06/2026	Rachid Zarouali	Contribution NBS
0.2	22/06/2026	Anne-Cécile Orgerie	Contribution CNRS-IRISA
1.0	26/06/2026	Thomas Cadeau, Damien Monguillon	Relecture et ajustements finaux

Table des matières

PROPRIÉTÉS DU DOCUMENT	1
HISTORIQUE DES CHANGEMENTS	1
TABLE DES MATIÈRES	3
LISTE DES TABLEAUX.....	4
LISTE DES FIGURES	4
1 INTRODUCTION.....	6
2 OBJECTIVES	7
3 TECHNICAL BACKGROUND ISSUES OF CONTINUUM EDGE-CLOUD	8
4 TECHNICAL ISSUES ADDRESSED	9
5 APPROCHES AND TECHNOLOGIES USED	10
6 USE CASES AND EXPECTED IMPACT.....	12
7 RESTITUTION OF CONTRIBUTORS' CONTRIBUTIONS.....	13
7.1 ATOS.....	13
7.1.1 Introduction.....	13
7.1.2 Work done.....	13
7.2 CGI FRANCE	17
7.2.1 Introduction.....	17
7.2.2 Challenges.....	17
7.2.3 Work Carried out.....	19
7.3 UT3-IRIT-SEPIA.....	32
7.3.1 Goal.....	32
7.3.2 State of the Art.....	32
7.3.3 Pwap Architecture and Implementation.....	33
7.3.4 Next Steps.....	34
7.4 CNRS-IRISA	34
7.4.1 Introduction.....	34
7.4.2 Challenges.....	35
7.4.3 Work Carried out.....	35
7.5 PROVENRUN.....	37
7.5.1 Introduction.....	37
7.5.2 Challenges.....	37
7.5.3 Work Carried out.....	37
7.6 ARMADILLO.....	37
7.6.1 Introduction.....	37
7.6.2 Challenges.....	37
7.6.3 Work Carried out.....	39
7.7 NINGALOO	41
7.7.1 Introduction.....	41

7.7.2	Challenges.....	41
7.7.3	Work Carried out.....	41
7.8	RYAX.....	42
7.8.1	Introduction.....	42
7.8.2	Challenges.....	42
7.8.3	Work Carried out.....	42
7.9	NBS.....	45
7.9.1	Introduction.....	45
7.9.2	Challenges.....	45
7.9.3	Work Carried out.....	46
8	CONCLUSION AND NEXT STEPS.....	47
9	ACRONYMES ET ABBREVIATIONS.....	48

Liste des tableaux

TABLE 1 : criteria.....	33
TABLE 2 : Acronyms and Abbreviations.....	37

Liste des figures

Figure 1 : High-level diagram of the Cloud-Edge Orchestration Solutions.....	6
Figure 2 : Testbed platform.....	10
Figure 3 : Principe of the Preemption Mechanism.....	14
Figure 4 : Example manifest showing the preemptive controller labels.....	20
Figure 5 : Execution Order of Pods of Deployments using our preemption controller.....	22
Figure 6 : Execution Order of Pods of Deployments using native Kubernetes preemption.....	23
Figure 7 : Execution Order of Pods standalone using our preemption controller.....	23
Figure 8 : Execution Order of Pods standalone using native Kubernetes preemption.....	24
Figure 9 : Principle of our work, which we plan to further develop as part of the research activities related to orchestration solutions.....	25
Figure 10 : CRIU Operating Stages.....	26
Figure 11 : Architecture of Pswap.....	27
Figure 12 : Architecture Ryax – site level view.....	35

1 INTRODUCTION

Deliverable 2.3 aims to ensure that the applications and services of the E2CC project can be orchestrated securely, efficiently, and sustainably across the entire Edge-Cloud continuum, while integrating the latest advances in monitoring, energy optimization, and security.

This deliverable is part of the project's second key milestone. The stakeholders involved are the following partners:

- Atos
- NBS System
- CGI France
- UT3-IRIT-SEPIA
- CNRS-IRISA
- UGA-LIG-ERODS
- ProvenRun
- Armadillo
- Ningaloo
- Ryax

2 OBJECTIVES

Deliverable 2.3 aims to develop and deliver orchestration solutions for the Cloud-Edge continuum, integrating three major focus areas:

- **Security:** ensure the protection of environments and applications deployed across the continuum.
- **Optimal resource management:** orchestrate the placement and execution of applications and services to maximize efficiency.
- **Energy efficiency:** minimize energy consumption and the carbon footprint of infrastructures and applications.

3 TECHNICAL BACKGROUND ISSUES OF CONTINUUM EDGE-CLOUD

The E2CC project is part of the digital transformation of IT infrastructures, where applications and services are no longer hosted solely in the cloud, but extend all the way to the edge of the network. This involves:

- **Heterogeneous resources** (servers, edge devices, networks, etc.)
- **Performance constraints** (latency, bandwidth, availability)
- **Increased security requirements** and access management
- A need for **energy efficiency** and a reduced carbon footprint

4 TECHNICAL ISSUES ADDRESSED

Deliverable 2.3 aims to address several major technical challenges:

- **Multi-level orchestration:** Deploy, manage, and supervise applications and services across hybrid infrastructures (Edge and Cloud), taking into account the diversity of environments and business needs.
- **Integrated security:** Ensure the security of data, communications, and access across the entire continuum, with mechanisms for isolation, monitoring, and automatic remediation.
- **Resource optimization:** Dynamically allocate resources (CPU, memory, storage, network) according to application requirements, energy constraints, and service quality policies (SLAs).
- **Energy management:** Integrate energy models to optimize application placement and reduce energy consumption, taking into account the availability of renewable energy and carbon impact.
- **Anomaly detection and management:** Use advanced techniques (AI, machine learning) to detect, anticipate, and correct failures or performance anomalies in distributed and complex environments.

5 APPROCHES AND TECHNOLOGIES USED

- **Cloud-native orchestration** (Kubernetes, microservices, containers) adapted to the Edge, with orchestrator federation and support for multiple lightweight operating systems.
- **Advanced monitoring:** Supervision tools covering the entire application lifecycle, from hardware to software, including the collection of energy and performance metrics.
- **Security by design:** Integration of security solutions from the design phase onward, including identity management, network segmentation, incident detection, and automatic remediation.
- **Standardization and interoperability:** Contribution to the evolution of standards (IoT, MEC, oneM2M) to ensure solution interoperability and long-term sustainability.
- **Multi-stakeholder collaboration:** Synergy between industrial partners, academia, and European partners to pool expertise and accelerate innovation.

The illustration below shows a high-level view of the various components that make up Work Package 2, broken down by functional layers. The components of this deliverable are located in the Orchestration layer.

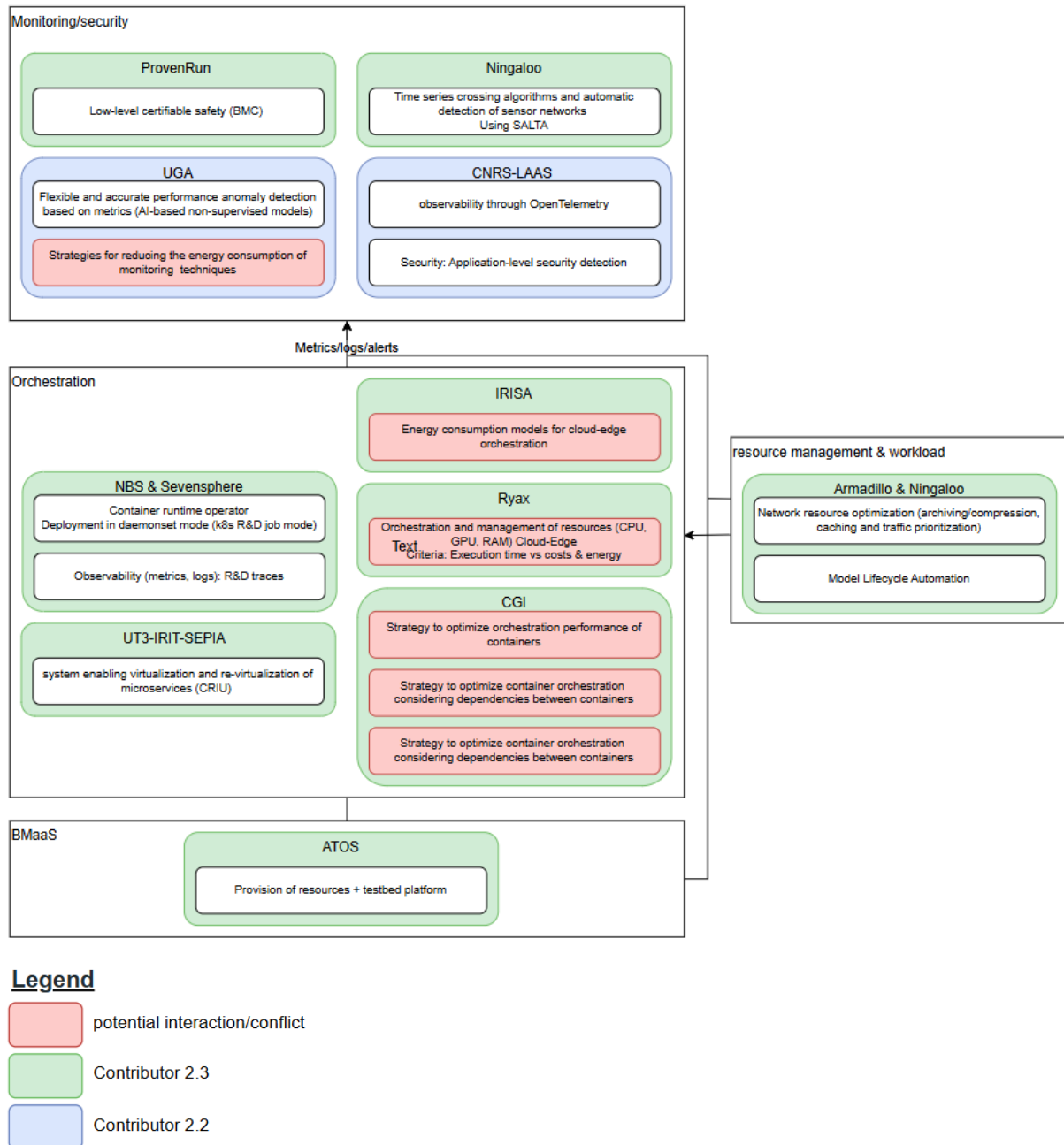


FIGURE 1 : HIGH-LEVEL DIAGRAM OF THE CLOUD-EDGE ORCHESTRATION SOLUTIONS

6 USE CASES AND EXPECTED IMPACT

- **Deployment of critical applications** (computer vision, predictive analytics, etc.) on hybrid infrastructures.
- **Reduction of energy consumption** and the carbon footprint of datacenters and Edge equipment.
- **Strengthening digital sovereignty** and the security of European infrastructures.

7 RESTITUTION OF CONTRIBUTORS' CONTRIBUTIONS

7.1 Atos

7.1.1 Introduction

The goal is to provide an orchestration software stack that will serve as the foundation for the installation of all the E2CC components. This stack will be installed on a Testbed platform (as part of Workpackage 7 task) to validate the integration of all E2CC components, including the WP2 components described in this report.

7.1.2 Work done

1. <https://neonephos.org/https://image-builder.sigs.k8s.io/>
The project adopts a layered approach to management, ensuring that both physical hardware and virtual clusters are treated as programmable resources.
2. Bare Metal & IaaS Provisioning: The foundation relies on ClusterAPI (CAPI) to provide a unified entry point. It treats infrastructure as code, allowing the team to swap or mix providers—Proxmox, IronCore, and KubeVirt—using the same declarative YAML manifests. From the Proxmox side it will facilitate the workflow by including image builder to build templates compatible with Kubernetes (<https://image-builder.sigs.k8s.io/>)
3. Metal Management: For physical servers, IronCore (NeoNephos) acts as the Bare Metal as a Service (BMaaS) engine. The E2CC team is actively developing custom extensions for IronCore to bridge the gap between raw hardware and Kubernetes automation. The usage of the redfish API is key for manageability and this limits hardware compatibility. Similarly, IPv6 seems to be the standard used in IronCore and future developments could be in that direction in order to make compatibility work with the e2cc platform.
4. Kubernetes Distributions: The platform standardizes on KubeAdm for standard deployments and K3s for lightweight or edge scenarios, ensuring a consistent runtime environment across different hardware profiles. However, multiple container runtimes could be needed at the same time, that is why KOS is also on the roadmap
5. Fleet Management and Federation
To avoid "cluster sprawl," the project uses Open Cluster Management (OCM). This acts as the central brain (the "Hub") that orchestrates multiple "Spoke" clusters
6. Unified Control: OCM allows administrators to push configurations, security policies, and workloads to any cluster regardless of where it is hosted (e.g., a Proxmox cluster in a data center vs. an IronCore bare-metal node).
7. Policy-Driven Governance: By integrating with Kyverno and Vault, OCM ensures that security and compliance rules are automatically inherited by every new cluster added to the fleet.

8. Continuous Deployment (GitOps)
The deployment strategy centers on GitOps, where the desired state of the entire platform is stored in Git. Probably the flexibility of ArgoCD vs. Flux will be preferred to have more control on how applications and platform components are synchronized.
9. 4. Observability and Environmental Impact
Orchestration is closely tied to the WorkPackage 3 goals of decarbonization. The stack goes beyond standard performance monitoring:
10. While some of the orchestration decisions can be at the platform level by scaling kubernetes clusters, the monitoring can also help to better orchestrate services accross clusters using open cluster management.
11. Energy Orchestration: The inclusion of Alumet and Mojito/S allows the orchestrator to monitor power consumption. This data can eventually influence "green scheduling," where workloads are moved to clusters with lower carbon intensity.
12. 5. Integrated Security Architecture
Security is baked into the orchestration layer rather than added on top.
13. Identity & Secrets: Keycloak handles authentication, while Vault and Cert-manager automate secret rotation and TLS certificates across the fleet.
14. Hardware-Rooted Trust: The integration of the Cosmian HSM and physical KMS servers (Trustway) ensures that encryption keys are managed with the highest level of physical security during validation tests.
15. The integration between the different solutions is still missing the final implementation details but this will be achieved over time when the platform is fully operational.

7.1.2.1 Testbed platform description

The diagram in the figure (next page) illustrates the hybrid Cloud & Edge infrastructure (provided by Packages 1 and 2, testbed tasks) designed for the integration of the E2CC platform software bricks and the tests performed by all partners, including use cases.

At the top of the diagram we see how the test platform interacts with the outside world. There will be a set of external services to which it will have to connect: public clouds (GCP) for the IAM service developed in batch 5, code repositories (GitHub), AI/ML platforms (HuggingFace, Kaggle, MLFlow) and container registries (DockerHub, Quay.io).

All traffic must pass through a firewall with a "whitelist" of IP addresses, ensuring that only authorized IPs can access the internal infrastructure. The "Bastion" is a Virtual Machine (VM) running (with other central services) on a Bull-S200 server (**in yellow**). The "Bastion" VM serves as an entry point for the entire E2CC test platform.

Administrators and automated systems use the "Bastion" VM that stores KubeConfig files to control the various Kubernetes clusters (K8s). The core of the platform is a Kubernetes system (K8s) with:

1. The management cluster (**in purple**) is a K8s cluster dedicated to administrative tasks, including bare metal provisioning via ClusterAPI.
2. Les clusters de « workload » :
3. K8s Cloud (**green**): Simulates cloud providers for centralized, resource-intensive tasks.
4. K8s Edge (**blue**): Simulates decentralized or edge-deployed environments in the field.
5. The rack (**in pink**) integrates the hardware dedicated to the security of batch 6 (the KMS server and the encrypted NAS)

In conclusion, here is a summary of the list of software/technologies/tools that will be installed and tested on our test platform: for orchestration, the Kubernetes distributions (K8s) considered first are at least KubeAdm and K3s. The K8s federation and the management of the various K8s Clusters will be provided by Open Cluster Management (OCM, <https://open-cluster-management.io/>). Infrastructure Management as a Service (IaaS) will use the ClusterAPI with possible service providers ProxMox, VMware, IronCore and KubeVirt to be tested. For the Bare Metal as a Service (BMaaS) part, NeoNephos' IronCore will be used with some extensions that we plan to develop to contribute to the project. The deployment should use the solution proposed by NeoNephos (ArgoCD, CRDs, and their operators) but Sylva's solution based on Flux and "Helm charts" remains under consideration. The observability of the E2CC platform will be based on the tools offered by Sylva (Prom, Loki, Grafana, etc.) with the addition of Opentelemetry, Thanos, Tempo, Forecasts, Alumet (<https://alumet.dev>) and Mojito/S (<https://gitlab.irit.fr/sepia-pub/mojitos>). The last 2 (Alumet and Mojito/S) being developed by E2CC for decarbonization (in WorkPackage 3). Security will be based on the tools offered by Sylva (Keycloak, Crossplane, Vault, Cert-manager, Kyverno) with all the tools provided by WorkPackage 6: The Cosmian HSM and the possibility of using a physical KMS server (such as the one provided by Trustway and which will be used for validation tests).

<https://open-cluster-management.io/><https://alumet.dev><https://gitlab.irit.fr/sepia-pub/mojitos>

7.1.2.2 Workstreams IPCEI-CIS

Workstream 2 (WS2) of the IPCEI-CIS programme, Cloud Edge Capabilities, is the main WS of the E2CC project. In this context, Atos participates in the meetings of this WS2, and in particular in the working groups (or clusters)

1. Cloud-Edge Management and Orchestration
2. Cross Cutting Security
3. Cross Cutting Sustainability

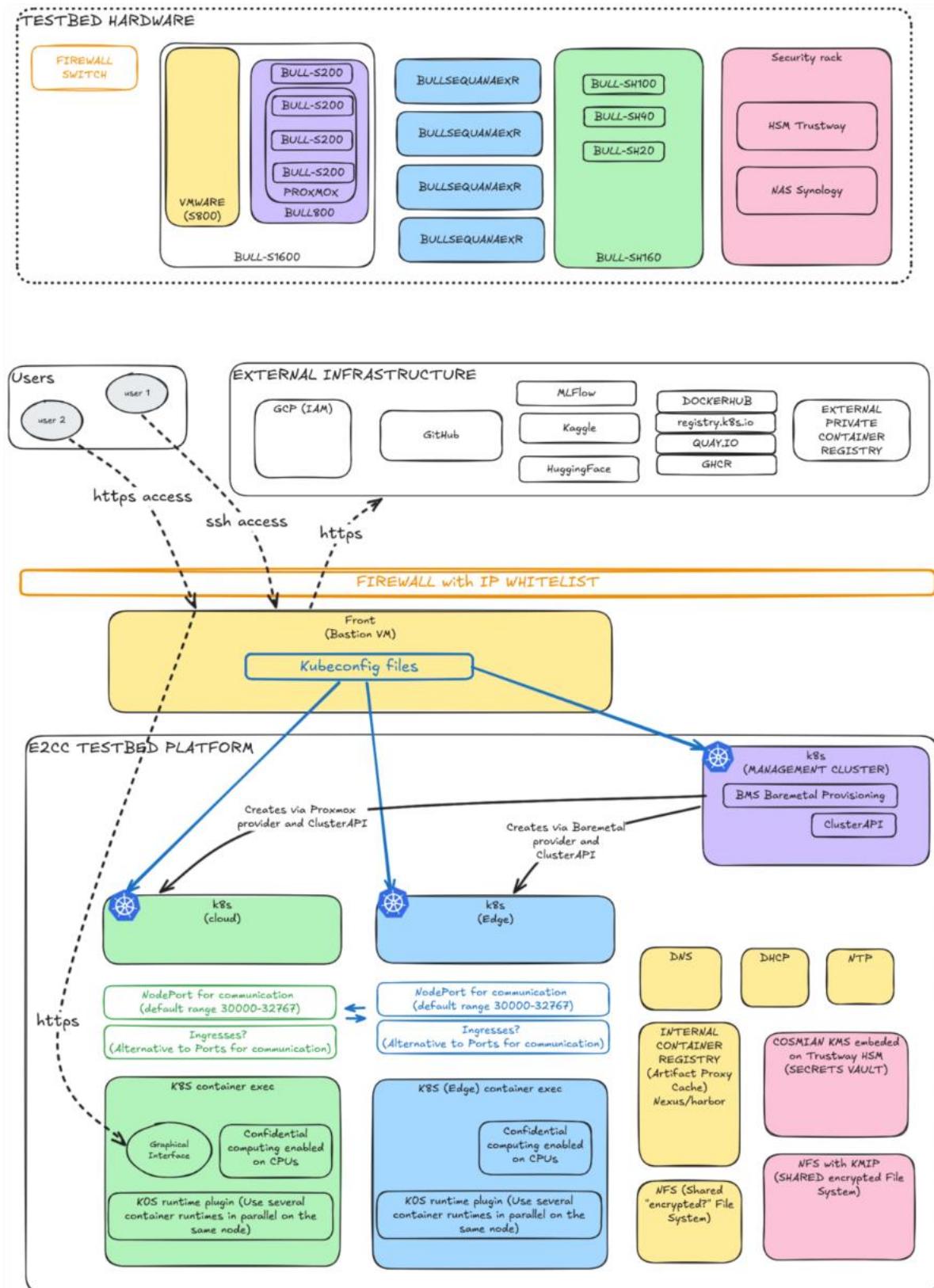


FIGURE 2 : TESTBED PLATFORM

7.2 CGI France

7.2.1 Introduction

Kubernetes has emerged as the leading platform for orchestrating containerized workloads across the cloud-to-edge continuum. However, its native scheduling and preemption mechanisms remain limited when handling crisis situations characterized by sudden resource shortages and the arrival of critical workloads. In such contexts, clusters may become saturated with lower-priority applications, preventing the scheduling of high-priority pods that are essential for maintaining system continuity or meeting urgent operational demands.

Our work focuses on crisis-aware resource management in Kubernetes environments. We propose a dynamic preemption Kubernetes orchestrator that acts as an external decision-making layer, activated specifically during resource crisis situations. A crisis situation occurs when critical high-priority pods remain pending because of insufficient available resources. Once such a condition is detected, the controller initiates a controlled preemption mechanism to reclaim resources from lower-priority workloads and ensure the rapid deployment of critical applications.

Unlike the default Kubernetes preemption mechanism, our approach integrates multi-criteria decision policies, such as resource consumption, execution progress, pod age, and replica redundancy, to minimize the impact of eviction while ensuring the rapid admission of critical workloads. Furthermore, the proposed solution supports reversible preemption, enabling evicted pods to be seamlessly restored once the crisis subsides, thereby preserving application integrity and service continuity. We emphasize that our approach is generic with respect to the number of criteria: criteria can be added or removed depending on user needs. The objective is to illustrate the core principle of our multi-criteria-based approach.

Experiments conducted with our intelligent crisis management solution for Kubernetes demonstrate significant improvements in scheduling responsiveness for critical workloads, more efficient resource reallocation under high-pressure conditions, and minimal disruption to lower-priority services.

7.2.2 Challenges

Kubernetes has become the de facto standard for orchestrating containerized workloads across heterogeneous environments spanning the cloud-to-edge continuum. Its scheduler provides advanced placement and scaling mechanisms, enabling clusters to efficiently run hundreds or thousands of applications. However, in situations of resource scarcity and sudden workload surges, referred to in this work as crisis situations, the default preemption model in Kubernetes exhibits significant limitations.

A crisis situation arises when the system faces resource saturation while critical workloads must be scheduled urgently. In such contexts, clusters may become locked by lower-priority applications, preventing the timely execution of high-priority pods that are essential for maintaining service continuity.

7.2.2.1 Limitations of Native Kubernetes Preemption

In the native Kubernetes scheduler, preemption is primarily priority-based: if a high-priority pod cannot be scheduled due to insufficient resources, the scheduler may evict lower-priority pods to make room. While functional, this mechanism is coarse-grained and considers only pod priority. It does not account for multiple factors such as resource consumption, pod age, computational progress, or replication level. As a result, indiscriminate eviction decisions during crisis situations can lead to excessive service disruption, wasted computation, and degraded application performance.

Without a more sophisticated and crisis-aware preemption strategy, the system may fail to react appropriately, delaying critical operations. Kubernetes' native preemption was primarily designed as a basic starvation-avoidance mechanism rather than a comprehensive crisis management solution.

Its main limitations include:

1. **Inflexible Priority Model:** Priority values are static and defined globally at pod creation time, limiting adaptability during crisis situations.
2. **Limited Observability and Control:** The scheduler operates as a monolithic component and does not expose detailed insights into preemption decisions. Implementing advanced or crisis-aware policies requires complex extensions such as custom scheduler plugins.
3. **Irreversible Eviction:** Evicted pods are permanently deleted. This behavior can lead to oscillations, instability, and inefficient resource usage, especially under crisis conditions.

These limitations highlight the need for a flexible, crisis-aware, and reversible preemption mechanism capable of adapting to dynamic workload conditions while minimizing system disruption..

7.2.2.2 Contributions

Our work introduces a crisis-aware approach to pod preemption with the following contributions:

1. **Flexible Crisis-Aware Priority Selection:** Extends the fixed-priority model by incorporating dynamic factors such as deadlines, external events or administrative policies.
2. **Enhanced Observability and Control:** Enables informed preemption decisions by evaluating multiple criteria and selecting the most suitable pods for eviction, thereby minimizing the impact on running services. In this work, we consider nine criteria, although the number can be adjusted as needed.
3. **Reversible Resource Reclamation:** Preempted pods are temporarily scaled down or evicted, with automated restoration once the crisis subsides, ensuring application consistency and stability.
4. **External Crisis Management Controller:** A decoupled mechanism operating independently from the Kubernetes scheduler, allowing seamless integration without modifying its core.
5. **Experimental Validation:** A comprehensive evaluation demonstrating reduced scheduling latency for critical workloads, improved resource utilization under crisis conditions, and safe restoration of preempted pods.

Together, these contributions provide a crisis-aware, flexible, and reversible preemption framework that addresses the limitations of the native Kubernetes scheduler and enables more resilient resource management.

7.2.3 Work Carried out

In the following, we present:

1. the operating principle of our preemptive controller mechanism,
2. an example illustrating the use of this controller,
3. the experiments conducted to validate our approach,
4. a summary highlighting the main outcomes of our work.
5. Finally, we discuss the future perspectives and the next developments that we plan to carry out.

7.2.3.1 Preemption Principle

The objective of our approach is to ensure that whenever a high-priority pod enters the *Pending* state, our preemptive controller selects intelligently the most appropriate lower priority pod(s) to preempt. This selection aims to maximize overall execution efficiency of pods in the Kubernetes cluster while minimizing the impact on running applications.

In the following, we describe:

1. the preemption principle,
2. the criteria considered in the preemption mechanism,
3. the multi-criteria decision-making algorithm applied for preemption.

Preemption Mechanism

Our preemption strategy is based on the algorithm presented in Algorithm 1.

Algorithm 1: Intelligent Pod Preemption algorithm

Input: Pending pod P_p
Input: List of running pods R
Output: Scheduled pending pod P_p with minimal impact on cluster and applications

```

foreach pending pod  $P_p$  in  $Pending$  do
    If  $P_p$  cannot be scheduled due to insufficient resources then
        Retrieve the priority  $prio_p$ , CPU  $cpu_p$ , and memory  $mem_p$  of  $P_p$ ;
        Identify candidate pods  $C$  in  $R$  such that:
            • Pod is currently running
            • Pod is preemptable (labeled accordingly)
            • Priority <  $prio_p$ 

        If  $C$  is empty then
            continue to next pending pod;
        Compute the nine criteria for each pod in  $C$ ;
        Apply the TOPSIS algorithm to rank pods and select those with minimal preemption impact;
        // Select pods to preempt to free sufficient resources
        Initialize  $to\_preempt = \emptyset$ ;
        foreach pod  $p$  in ranked  $C$  do
            Add  $p$  to  $to\_preempt$ ;
            Compute total CPU and memory freed by preempting  $to\_preempt$ ;
            If freed resources + available node resources  $\geq$  requirements of  $P_p$  then
                break;

        If resources remain insufficient then
            continue to next pending pod;
        // Preempt selected pods
        foreach pod  $p$  in  $to\_preempt$  do
            If  $p$  is controlled by a Deployment or StatefulSet then
                Scale controller replicas to 0 and store original replica count;
            else
                Delete standalone pod and store its specification for later restoration;

        Schedule pending pod and wait until it is Running and Ready
        Restore controllers to their original replica counts;
        Restore standalone pods to their original nodes;
    
```

This algorithm ensures that preemption decisions are made systematically using multi-criteria evaluation, balancing the need to schedule high-priority pods with minimal disruption to the cluster and the running workloads.

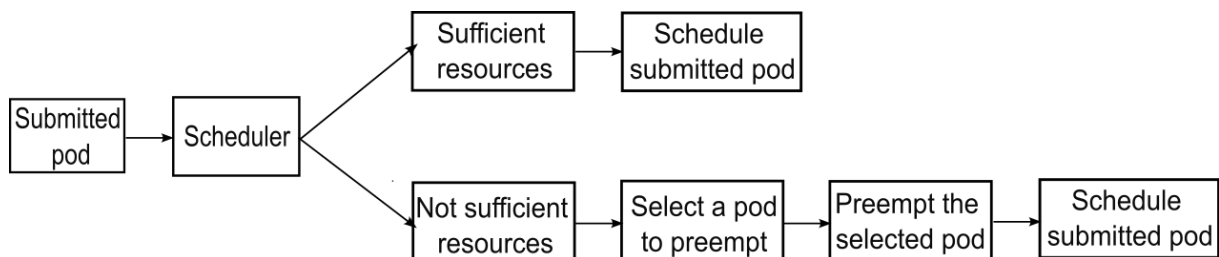


FIGURE 3 : PRINCIPE OF THE PREEMPTION MECHANISM

Figure 1 illustrates the principle of our preemption approach.

When a pod is submitted, the scheduler first checks if sufficient resources are available.

- If resources are available, the submitted pod is scheduled immediately.
- Otherwise, the scheduler looks for lower-priority pods that can be preempted. If such pods exist:
 - it selects the most suitable one for preemption.
 - the selected pod is then preempted.
 - finally, the higher-priority submitted pod is scheduled on the freed resources.

• Multi-Criteria Pod Selection for Preemption

In our approach, the selection of the pod to be preempted is formulated as a *Multi-Criteria Decision-Making problem*. The objective is to minimize the negative impact of preemption on both the cluster infrastructure and the running applications.

To achieve this goal, we consider nine criteria that capture different aspects of the cost and impact of preempting a pod. These criteria are classified into minimization and maximization objectives, as described below. Our approach is generic: although we consider nine criteria in this work, they can be adjusted by adding or removing criteria depending on user requirements. The objective is to illustrate our generic approach based on a multi-criteria decision framework.

Minimization criteria related to:

- Startup Time,
- Downtime,
- Restart Count,
- CPU Requested,
- Memory Requested,
- Checkpoint Distance,

Maximization criteria related to :

- Remaining Time,
- Age,
- Replicas Available.

In the following, we justify the rationale behind minimizing or maximizing each of these criteria.

Startup Time (Minimization)

The startup time corresponds to the time required to fully restart a pod after an interruption.

Preempting a pod with a high startup time may result in:

- a significant increase in recovery time,
- prolonged service unavailability,
- degradation of Service Level Objectives (SLOs).

Therefore, it is preferable to select pods with low startup times, as they can be redeployed more quickly after preemption. The objective is to minimize the restart overhead introduced by preemption.

Downtime (Minimization)

The downtime represents the service interruption caused by stopping a pod.

Preempting a pod that leads to high downtime may negatively affect:

- service availability,
- user experience,
- application-level Service Level Agreements (SLAs).

The objective is to select pods with low expected downtime to minimize the impact of preemption on system availability.

Restart Count (Minimization)

The restart count indicates the number of times a pod has been restarted during its lifecycle.

A high restart count may indicate:

- application instability,
- degraded runtime behavior,
- sensitivity to interruptions.

Preempting an already unstable pod could exacerbate system instability. Therefore, pods with low restart counts are preferred to avoid further destabilizing the application.

CPU Requested (Minimization)

The CPU requested represents the amount of CPU resources reserved by the pod.

Preempting pods with high CPU requirements may lead to:

- a significant loss of ongoing computational resources,
- wasted CPU cycles that have already been allocated and used.

Selecting pods with lower CPU requirements reduces the computational loss associated with preemption and limits its impact on overall cluster scheduling. Furthermore, lightweight pods are typically less constrained in terms of placement, making them easier to reschedule.

Memory Requested (Minimization)

Memory is a critical resource in Kubernetes clusters, and memory-intensive pods are often harder to reschedule after preemption.

Preempting a pod with a large memory footprint may lead to:

- substantial loss of in-memory state,

- high data reloading costs,
- performance degradation.

Selecting pods with lower memory consumption helps reduce memory fragmentation and decreases the risk of rescheduling failures. The objective is therefore to reduce the cost of reconstructing the application memory state.

Checkpoint Distance (Minimization)

The checkpoint distance represents the time or computational progress since the last checkpoint was saved.

If a pod is preempted far from its last checkpoint:

- a large amount of computational work may be lost,
- the application may need to recompute previously completed tasks.

Minimizing this distance reduces the amount of lost progress when preemption occurs. The goal is therefore to minimize the loss of computational progress.

Remaining Time (Maximization)

The remaining time represents the estimated time before the completion of the pod's execution.

Preempting a pod that is close to finishing its task may lead to:

- disproportionate computational loss,
- wasted progress that was nearly completed.

By maximizing the remaining execution time, the system favors preempting pods that still have a long execution horizon, thereby reducing the relative loss of useful work.

Age (Maximization)

The age of a pod represents the time elapsed since its startup.

Older pods often exhibit the following characteristics:

- they may have already produced partial outputs,
- their execution state may already be stabilized,

they may benefit from checkpointing or replication mechanisms.

Preempting older pods can sometimes allow for more efficient resource reallocation while avoiding unnecessary disruptions to recently started pods.

Replicas Available (Maximization)

The number of available replicas reflects the redundancy of the service.

If multiple replicas are running:

- stopping one pod will have limited impact on service availability,
- the application can continue operating normally.

Conversely, preempting a pod belonging to a service with few replicas may significantly degrade service performance or availability.

Therefore, it is preferable to preempt pods belonging to highly replicated workloads, thereby preserving service continuity.

• Preemption Modeling Using Multi-Criteria Analysis

our model integrates a set of criteria that collectively characterize the impact of preempting a running pod:

- restart overhead, encompassing startup time, downtime, and restart count,
- loss of computational progress, characterized by checkpoint distance and remaining time,
- resource consumption, reflected by CPU and memory requests,
- impact on service availability, captured through the number of available replicas and the age.

Combining these criteria allows the system to evaluate the overall impact of preempting each candidate pod. To determine the pod that provides the best trade-off among these criteria, we employ a multi-criteria decision-making (MCDM) approach based on the TOPSIS algorithm.

• Pod Selection Using the TOPSIS Algorithm

To identify the most suitable pod to preempt, we employ the *Technique for Order Preference by Similarity to Ideal Solution* (TOPSIS), a widely used method in multi-criteria decision-making. The nine previously defined criteria are all incorporated into our selection process.

The main idea of TOPSIS is that the optimal alternative should have:

- the shortest distance from the ideal solution,
- the largest distance from the negative ideal solution.

In our context, each candidate pod is represented as a vector composed of the nine criteria described above.

Step 1: Construction of the Decision Matrix

A decision matrix X is constructed where each row corresponds to a candidate pod and each column corresponds to a criterion:

$$X = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \dots & x_{mn} \end{bmatrix}$$

where m represents the number of candidate pods and n the number of criteria.

Step 2: Normalization of the Decision Matrix

Since the criteria may have different scales, the matrix is normalized using vector normalization:

$$r_{ij} = \frac{x_{ij}}{\sqrt{\sum_{i=1}^m x_{ij}^2}}$$

Step 3: Weighted Normalized Matrix

Each criterion can be assigned a weight reflecting its relative importance:

Where w_j represents the weight of criterion j .

Step 4: Determination of Ideal Solutions

Two reference solutions are defined:

- Positive ideal solution (best values),
- Negative ideal solution (worst values)

These solutions are determined depending on whether each criterion should be minimized or maximized.

Step 5: Distance Computation

For each candidate pod, the Euclidean distance to the ideal solutions is computed as:

$$D_i^+ = \sqrt{\sum_{j=1}^n (v_{ij} - v_j^+)^2}$$

$$D_i^- = \sqrt{\sum_{j=1}^n (v_{ij} - v_j^-)^2}$$

Step 6: Closeness Coefficient

Finally, a closeness score is computed for each pod:

$$C_i = \frac{D_i^-}{D_i^+ + D_i^-}$$

The pod with the highest score is considered the best candidate for preemption, as it represents the optimal compromise among all criteria.

7.2.3.2 Example of Using Our Preemptive Controller

To use our preemptive controller, two labels must be defined in each Kubernetes manifest:

preemption: "true" indicates that the pod is eligible for the preemption mechanism. This allows the scheduler (or a custom controller) to preempt lower-priority pods and allocate resources to this pod when node resources are constrained.

preemption-value: "100" defines the pod's priority level; higher values increase the likelihood of preempting other pods. In this example, the value is set to 100, but it can be adjusted depending on the criticality of the pod.

In this setup, labels are used to signal preemption to a custom controller. In contrast, native Kubernetes preemption relies on PriorityClass and the priorityClassName field. This label-based approach provides flexible and dynamic priority management directly through pod metadata.

The preemptive controller ensures that the pod can obtain the necessary resources even under high node load by temporarily evicting lower-priority pods if required.

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: pod-a-deployment
  labels:
    app: pod-a
spec:
  replicas: 1
  selector:
    matchLabels:
      app: pod-a
  template:
    metadata:
      labels:
        app: pod-a
        preemption: "true"
        preemption-value: "100"
    spec:
      containers:
      - name: app
        image: nginx
        resources:
          requests:
            cpu: "1"
          limits:
            cpu: "2"

```

FIGURE 4 : EXAMPLE MANIFEST SHOWING THE PREEMPTIVE CONTROLLER LABELS

Figure 4 illustrates an example of how the preemptive controller is defined in a Kubernetes manifest.

7.2.3.3 Experiments

In flowing, we present experiment with managed Pods and standalone Pod.

- **Managed Pods**

In the following, we present an experimental workflow that defines a set of five Kubernetes deployments (Deploy-A, Deploy-B, Deploy-C, Deploy-D, and Deploy-E), where the preemption behavior is controlled using custom labels. Each deployment runs a single replica of an Nginx container and specifies explicit CPU resource requests and limits.

Deployments A, B, and C request 2 CPUs (2000 millicores), Deployment D requests 1 CPU, and Deployment E requests 5 CPUs. These heterogeneous resource requirements enable the evaluation of scheduling behavior under constrained resource conditions. A delay of 10 seconds is introduced between the submission of each deployment.

To emulate different application startup characteristics, each Deployment defines startup, readiness, and liveness probes with distinct configurations.

The Deployments are configured with varying startup and readiness parameters to model different initialization behaviors. Deploy-A represents a fast-starting service, with a low startup probe failure threshold (3), short probing intervals (2 seconds), and a minimal readiness delay (2 seconds). Deploy-B introduces moderate startup latency by increasing the failure threshold to 15 and a longer readiness delay (10 seconds). In contrast, Deployments C, D, and E simulate slow-starting applications, with high startup probe thresholds (30) and extended readiness delays (20 seconds).

Overall, this configuration models services with progressively increasing startup times, from fast (Deploy-A) to significantly delayed initialization (Deployments C-E).

Regarding liveness behavior, the Deployments exhibit different failure detection dynamics. Deploy-A includes a liveness probe with a 5-second initial delay and a 5-second probing period (failure threshold of 3), enabling fast failure detection. Deploy-B includes a liveness probe with a 10-second initial delay and a 10-second probing period, enabling moderately fast failure detection. Deployments C, D, and E adopt more relaxed liveness configurations, with a 15-second initial delay and a 15-second probing period, resulting in slower detection of failures and potentially increased downtime.

This setup allows us to capture a range of failure detection behaviors, from highly responsive (Deploy-A) to more tolerant but delayed reactions (Deployments C-E).

Preemption policies are expressed through two custom labels: *preemption=true* and *preemption-value*. Deploys A, B, and C are assigned a preemption value of 100, Deployment D is assigned a value of 200, and Deployment E is assigned the highest value of 300. These values represent the relative importance of the workloads and are used to guide a custom preemption mechanism under resource contention.

pod-a-deployment-6dbc5b6c45-8bdnb	0/1	Pending	0	23s
pod-b-deployment-6fb5d8f4f4-ckkbr	1/1	Running	0	71s
pod-c-deployment-6fbc4d459-54hkw	1/1	Running	0	61s
pod-d-deployment-776fdc67cc-9j9f4	1/1	Running	0	51s
pod-e-deployment-65b6b5465f-zc45g	1/1	Running	0	40s

FIGURE 5 : EXECUTION ORDER OF PODS OF DEPLOYMENTS USING OUR PREEMPTION CONTROLLER

Figure 5 shows the order of execution of the Deployments. We observe that our preemption controller selects Pod of the Deploy-A to be preempted. When Deploy-E is submitted with a high priority value of 300 and Pods of Deployments A, B, C, and D are already running. Among them, Deployments A, B, and C have the lowest priority value (100), compared to Deployment D, which has a higher priority value of 200.

Therefore, the preemption decision is restricted to Pods of Deployments A, B, and C. Among these candidates, Pod of Deployment A is selected because it minimizes both the startup time and the downtime criteria. Specifically, Deployment A is configured with the shortest

startup probe configuration and the most aggressive failure detection settings, resulting in faster recovery and lower preemption cost.

This result highlights the effectiveness of the multi-criteria decision mechanism, which goes beyond simple priority comparison by incorporating runtime characteristics such as startup behavior and failure detection dynamics.

Our Preemption Controller vs. Native Kubernetes Preemption

In this section, we implements a similar experimental scenario but relies on the native Kubernetes PriorityClass mechanism instead of custom preemption labels. The same five deployments are used, maintaining identical container images, resource requirements, and health probe configurations.

Deployments A, B, and C are assigned the low-priority class, Deployment D is assigned the average-priority class, and Deployment E is assigned the high-priority class. This configuration leverages Kubernetes' built-in priority and preemption mechanism, allowing higher-priority Deployments to preempt lower-priority ones when cluster resources are insufficient.

The startup, readiness, and liveness probes remain identical to those in the first workflow. This allows the comparison to focus exclusively on the difference between the custom label-based preemption approach and the native Kubernetes priority-based scheduling mechanism.

Similarly to the first workflow, deployments are submitted sequentially with a 10-second delay between each submission.

pod-a-deployment-64bcc5cdc7-pckp9	1/1	Running	0	65s
pod-b-deployment-6c69cf8796-kbls7	1/1	Running	0	55s
pod-c-deployment-5f954ff6f9-x4f6k	0/1	Pending	0	24s
pod-d-deployment-68b86cfdc5-ght7w	1/1	Running	0	34s
pod-e-deployment-6db6fd9b75-fwskp	1/1	Running	0	24s

FIGURE 6 : EXECUTION ORDER OF PODS OF DEPLOYMENTS USING NATIVE KUBERNETES PREEMPTION

Figure 6 shows the order of execution of the pods. We observe that the native Kubernetes preemption mechanism selects pod of Deployment-C for preemption. However, as illustrated in Figure 5, our controller instead selects Pod of Deployment-A for preemption.

Pod of Deployment-A is the most relevant candidate to preempt because it minimizes the overall preemption cost according to our multi-criteria model. In particular, it exhibits the lowest startup time and downtime compared to Pods of Deployments B and C, meaning that preempting it results in faster recovery and reduced service disruption.

In contrast, the native Kubernetes preemption mechanism relies primarily on priority. As a result, it may select a suboptimal pod (Pod-C in this case), leading to higher recovery time and increased impact on the system.

This comparison highlights the advantage of our approach, which incorporates additional operational metrics to make more informed and efficient preemption decisions.

- **Standalone Pod**

The same workflow used for managed pods is now deployed as standalone pods. This allows us to evaluate the behavior of the system without the abstraction layer introduced by higher-level controllers such as Deployments. By doing so, we can directly observe the impact of the preemption strategy on individual pods, ensuring that the results are not influenced by replica management or controller-driven rescheduling mechanisms.

pod-a	0/1	Pending	0	24s
pod-b	1/1	Running	0	71s
pod-c	1/1	Running	0	60s
pod-d	1/1	Running	0	50s
pod-e	1/1	Running	0	40s

FIGURE 7 : EXECUTION ORDER OF PODS STANDALONE USING OUR PREEMPTION CONTROLLER

Figure 7 shows that, using our controller, Pod-A is placed in a standby state to allow the execution of Pod-E, which has a higher priority. Instead of being immediately terminated, Pod-A is gracefully preempted, thereby reducing the overall disruption and enabling a more efficient use of cluster resources. When resources become available, the pod can be scheduled.

This behavior demonstrates the advantage of our approach, which considers both priority and runtime characteristics, allowing a more controlled and cost-aware preemption compared to the default Kubernetes mechanism.

Our Preemption Controller vs. Native Kubernetes Preemption

we execute in this section the same pod executed in the previous section.

pod-a	1/1	Running	0	68s
pod-b	1/1	Running	0	57s
pod-d	1/1	Running	0	37s
pod-e	1/1	Running	0	26s

FIGURE 8 : EXECUTION ORDER OF PODS STANDALONE USING NATIVE KUBERNETES PREEMPTION

Figure 8 shows that, under the native Kubernetes preemption mechanism, pods are directly evicted when higher-priority workloads need to be scheduled. In this case, Pod-C is evicted to make room for the execution of Pod-E.

However, compared to our preemption controller, this approach leads to a more disruptive behavior. In our approach, instead of being terminated, the selected pod is placed in a standby state. When resources become available again, the standby pod can resume execution without requiring a full restart.

This mechanism reduces the overhead associated with pod eviction and recreation, and improves overall system efficiency by preserving the execution state of preempted workloads.

7.2.3.4 Summary

In our work, we proposed a dynamic and reversible preemption controller for Kubernetes, designed to overcome the limitations of the native mechanism during crisis situations, defined here as the inability to schedule high-priority Pods due to insufficient available resources.

Unlike the standard approach, our solution introduces an external decision-making layer based on multi-criteria policies that consider several factors, including resource consumption, execution progress, Pod age, and replica redundancy. This strategy reduces the impact on running services while enabling the rapid admission of critical workloads. In addition, the reversibility of the mechanism ensures the restoration of preempted Pods once the situation has stabilized.

7.2.3.5 Perspectives

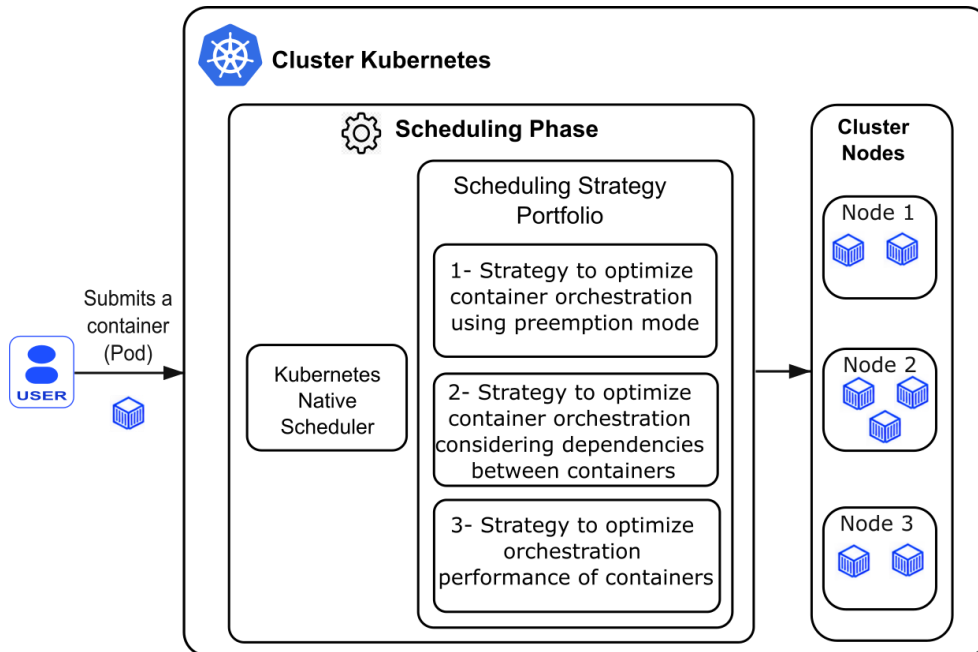


FIGURE 9 : PRINCIPLE OF OUR WORK, WHICH WE PLAN TO FURTHER DEVELOP AS PART OF THE RESEARCH ACTIVITIES RELATED TO ORCHESTRATION SOLUTIONS.

Figure 9 illustrates the overall principle our work, which we plan to further develop as part of the research activities related to orchestration solutions. When a user submits a container (pod), the Kubernetes scheduler analyzes its characteristics and selects the most appropriate scheduling strategy. If the user does not request one of the strategies included in our scheduling strategy portfolio, the native Kubernetes scheduler is applied by default. However, when a specific strategy is selected, the corresponding mechanism is used to identify the most suitable node in the cluster for efficient container deployment.

In this deliverable, we presented our preemptive scheduling strategy. As part of our future work, we plan to develop:

- a scheduling strategy that considers dependencies between containers,
- a scheduling strategy dedicated to optimizing orchestration performance.

7.3 UT3-IRIT-SEPIA

7.3.1 Goal

The goal is to design a solution that enables the migration of a microservice from a container to a virtual machine (VM) and vice versa, while minimizing resource consumption. The work carried out during this first year is organized around three main stages: a review of the existing state-of-the-art solutions, the design of our solution's architecture, and finally the initial implementation phase.

7.3.2 State of the Art

The first phase consisted of a review of existing service migration techniques. Three works were particularly noteworthy:

Catelyzer. Proposes an ultra-fast startup serverless sandbox system, relying on restoration from a pre-formed checkpoint rather than a cold start. Although the context differs from ours, their approach assumes a well-defined state that can be shared across workloads. The on-demand restoration technique remains an interesting avenue to explore for service migration.

Portkey. Focuses on the migration of containers nested within VMs, with the objective of reducing CPU consumption related to network operations by up to 75–82%. Although our priority is more latency than CPU consumption, the data transfer approach between the VM and the hypervisor presented in this work is very interesting for our research.

mWarp. Proposes live migration of containers from VM to VM on the same physical host through a highly efficient memory remapping mechanism that avoids massive page copies. This work is the closest to ours: the main difference is that we seek to migrate not from VM to VM, but directly from the VM to the host. It therefore constitutes an interesting reference.

At the end of this study, **CRIU (Checkpoint/Restore In Userspace)** emerged as the reference tool for application migration. It is in fact used in the majority of the works surveyed (Portkey, mWarp). CRIU makes it possible to capture the complete state of a running process into image files and then restore this process identically on another platform. Its operation is divided into two phases: checkpointing (freezing and serializing the state of the process) and restoration (reloading the process from the image files). Figure IRIT-1 presents the two stages of CRIU. The analysis of CRIU made it possible to identify two types of captured states: application states (virtual memory used by the application, by far the largest component) and kernel states (kernel control structures, much lighter). In the context of our work, we focus on application states, which are necessary to perform service migration.

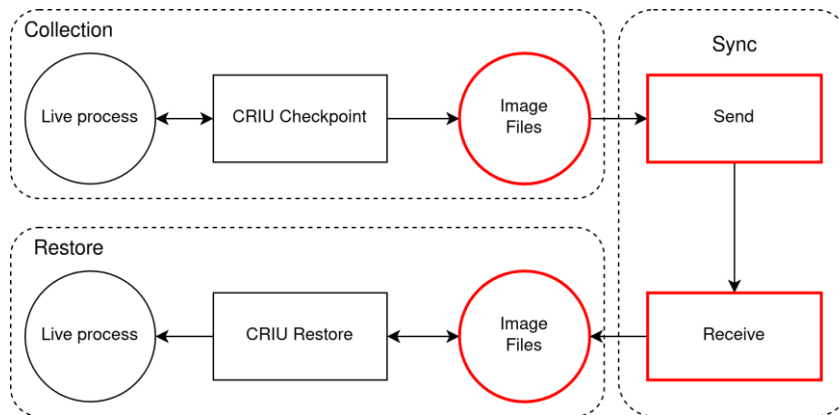


FIGURE 10 : CRIU OPERATING STAGES

7.3.3 Pwap Architecture and Implementation

Based on this understanding, an initial implementation was started on the Xen hypervisor around an architecture called **Pswap**. The architecture we developed (illustrated in Figure IRIT-2) consists of three modules: a guest module, a hypervisor module (IO module), and a host module. These modules collaborate in six steps to move the memory pages of a process from the VM to a ramfs mount point on the host.

The workflow is as follows: modified CRIU sends the virtual addresses of the pages to be migrated to the guest module, which translates them into physical addresses and forwards them to the hypervisor module via a hypercall. The latter opens the target files in the ramfs and delegates to the host module the extraction and transfer of the pages directly into the page cache of the destination files. CRIU can then restore the process on the fly from the ramfs.

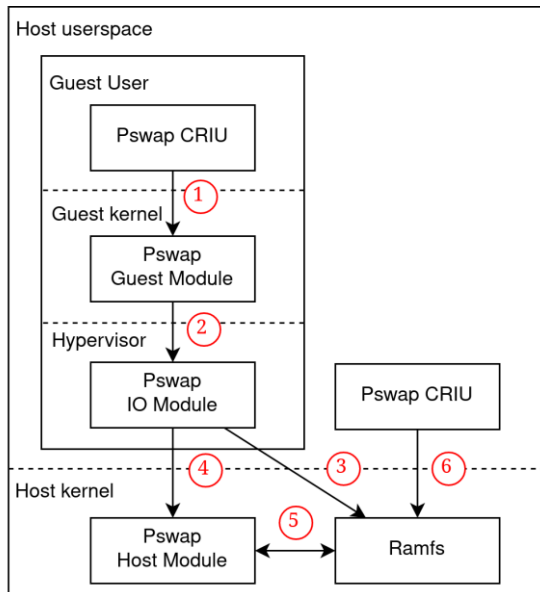


FIGURE 11 : ARCHITECTURE OF PSWAP.

The central technical choice of this implementation is **zero-copy**: rather than copying data from memory to an intermediate buffer and then to the destination file, the pages are transferred directly into the page cache of the target file, thereby avoiding any unnecessary copying. This approach aims to minimize resource usage and migration latency, which is the primary objective of this work. We have begun implementing our prototype.

7.3.4 Next Steps

For the next steps, we will continue the implementation of the prototype and set up a testing environment to evaluate our solution. This evaluation phase will allow us to measure the performance of the proposed approach, particularly in terms of migration latency and resource consumption. The objective is to achieve an efficient microservice migration mechanism while minimizing the overhead introduced during the migration process. We will also conduct a comparative study with existing solutions in order to assess the advantages and limitations of our approach and validate the effectiveness of the proposed architecture.

7.4 CNRS-IRISA

7.4.1 Introduction

Deploying applications on edge continuum infrastructures requires the use of various hardware and software components distributed across the infrastructure. These components are heterogeneous, and their power consumption depends partly on their usage. Therefore, several options exist for deploying applications on these distributed infrastructures.

To be energy efficient, application deployment strategies must rely on reliable end-to-end energy models to find feasible trade-offs between application performance (e.g., latency) and energy consumption. These end-to-end energy models should consider the electricity used by the different components during application execution, as well as a portion of the electricity used by the facilities hosting the components (e.g., electricity used for cooling). The objective of our work is to propose an energy consumption model for edge and cloud computing infrastructures (including networking devices in-between) during its usage phase. These models will be applicable to the edge-to-cloud continuum, taking into account, in particular, the equipment used for cooling. The models developed within this framework will allow for the evaluation of the energy efficiency of existing edge-to-cloud continuum infrastructures. They will also ultimately enable the development of environmental impact models for edge and cloud infrastructures, as well as effective multi-criteria optimization techniques for the deployment of distributed applications.

7.4.2 Challenges

Energy monitoring of distributed systems involves measuring the power consumption of servers and all data center components (network equipment, storage arrays, cooling systems, etc.). These measurements can be performed using either hardware wattmeters (additional equipment) or software wattmeters that utilize hardware counters present in modern processor architectures. Hardware wattmeters offer the advantage of providing accurate measurements for entire servers but are expensive, while software wattmeters are inexpensive to implement and allow for estimates per process running on a server, but they require energy consumption allocation models based on resource usage. In an edge-cloud context, virtualization layers add further complexity to allocation models. Whether obtained via hardware or software power meters, these measurements serve as the basis for calculating the carbon footprint of Cloud infrastructures and the carbon footprint per user of these shared platforms. Software power meters in the literature use attribution models that are primarily based on CPU and GPU usage. However, CPUs and GPUs are not the only resources used, and other resources would also be relevant if considered by the attribution models (network, disks, memory, etc.).

7.4.3 Work Carried out

The work carried out focuses on the study of energy models for the edge-cloud continuum. A review of the existing literature led to the identification of major approaches (based on resource usage or hardware counter) and the discovery of gaps in the literature. A proposed

model, incorporating elements from the literature and introducing new ones, is currently being developed. In particular, we identified that current software power meters base their energy allocation decisions solely on CPU and GPU usage, without considering the use of other resources (memory, disk, network). We studied the various available hardware counters that could account for the use of these other resources and we are currently proposing a new model that will be evaluated on the SLICES-FR experimental platform, notably by comparing it to the approach developed by Alumet.

The following figure is a motivational example of our work. It shows per-process energy attribution on an Intel processor of the SLICES-FR platform through Scaphandre (software watt-meter) with a 1 second measurement step. Energy consumption includes both Intel RAPL domains (package and DRAM). The scenario includes two different types of stress workloads: first, a compute intensive one (matrix product using 4 threads) and second, a memory intensive one (4 streams).

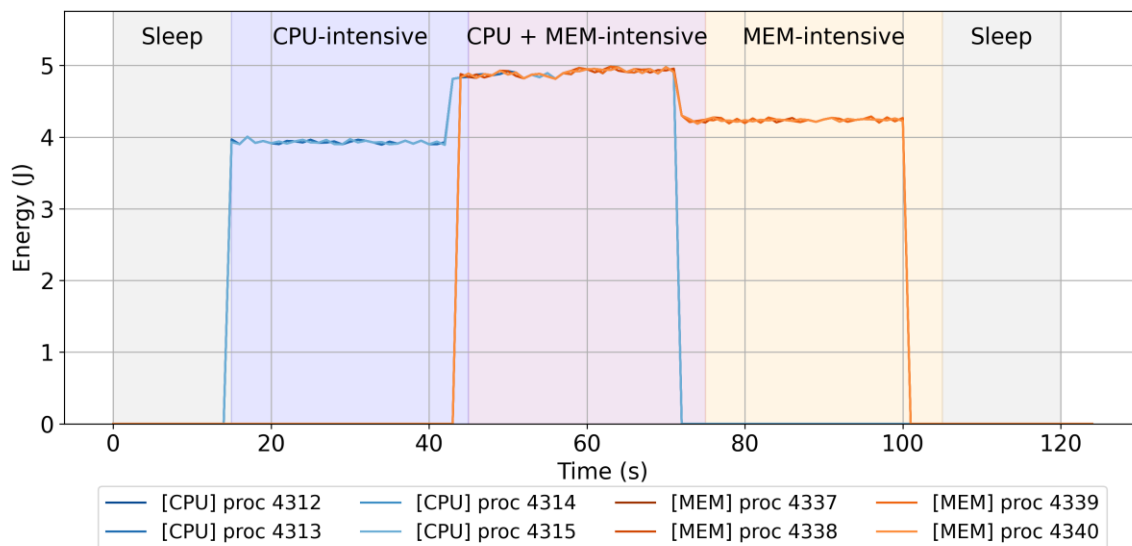


FIGURE 12 : PER-PROCESS ENERGY ATTRIBUTION ON AN INTEL PROCESSOR THROUGH SCAPHANDRE WITH A 1 SECOND MEASUREMENT STEP. ENERGY CONSUMPTION INCLUDES BOTH INTEL RAPL DOMAINS (PACKAGE AND DRAM).

The scenario is like follow: starting with a sleep for 15 seconds then running the first stress workload for 30 second alone, and then having both of the different workloads running simultaneously for 30 seconds, then stopping the first one keeping the memory intensive one for 30 second and finally a 15 seconds sleep time. It is obvious on the figure that the energy consumption while having both workloads running at the same time (violet phase) is not the sum of the energy consumption of the two workloads independently (blue and yellow phases). Also, when doing a simple allocation policy that divides 'fairly' the power consumption among the processors, it results in a process being exactly the same for 30 seconds but having a different energy consumption for the first 15 seconds than for the last 15 ones. We propose to design a fair allocation policy for energy consumption among processes that is based on hardware counters.

To identify relevant hardware counters, we conducted an experimental campaign on SLICES-FR on different servers with varying software loads (based on benchmarks). Analyzing their energy traces combined with data from various hardware counters allowed us to identify the most relevant counters. Indeed, accessing hardware counters is costly and limited, making it essential to identify the most pertinent ones and rely solely on them for regular access. Simultaneously, our review of existing literature models has led us to propose a unified model for converting the energy consumption of an edge-cloud infrastructure into a carbon footprint. This work is currently ongoing (redaction phase and evaluation phase led in parallel).



7.5 ProvenRun

7.5.1 Introduction

7.5.2 Challenges

7.5.3 Work Carried out

7.6 Armadillo

7.6.1 Introduction

In the E2CC platform architecture defined last year, our work integrates into the **"AI Capabilities"** layer of the **"E2CC Services"** layer. In work package 3, we are developing AI algorithms and models that leverage both edge and cloud data, involving exchanges in terms of memory, computing, networking, and security between the E2CC platform and the various tasks required to build and deploy the models.

Our focus is on **Cloud-Edge orchestration solutions** for resource management, security, and energy efficiency, as part of our applied work on developing post-processing algorithms for data inferred by neural networks. These algorithms enable the use of generic models and adapt them to specific tag hierarchies (thesaurus) for indexing and documentary search. The solution is based on an **online learning model**.

In this context, **Cloud-Edge orchestration must balance performance, scalability, and energy efficiency**. While cloud computing is powerful, it has limitations in terms of latency, bandwidth, and energy costs, especially when dealing with large datasets or tasks requiring immediate responsiveness. Conversely, processing data at the edge brings computation closer to the data source, reducing network transfers and improving efficiency. However, its deployment raises challenges in node coordination, resource management (CPU, GPU, storage), and energy optimization.

Our work focuses on **global infrastructure optimization**, combining cloud and Edge to:

1. Reduce network load through compression, caching, and traffic prioritization.
2. Minimize latency via local data processing and intelligent resource allocation.
3. Ensure energy efficiency by optimizing Edge device usage and automating the AI model lifecycle.

7.6.2 Challenges

In an ecosystem where data and AI models are increasingly resource-intensive, several major challenges arise for effective Cloud-Edge orchestration:

7.6.2.1 Resource Optimization and Network Footprint Reduction

1. How can we limit data transfer while preserving integrity and accessibility?
AI models, some of which can reach several GB in size, require frequent transfers between cloud and Edge nodes. Our solutions (compression, archiving) provide partial gains, but their effectiveness depends on a global strategy including data selection, preprocessing, and caching. The goal is to avoid network saturation while ensuring real-time access to critical data (models, inference results).
2. How can we balance load between cloud and Edge to avoid bottlenecks?
Processing data at the edge reduces latency, but its effectiveness depends on the capacity of local devices (CPU, GPU, storage) and their coordination with the cloud. Poor task distribution can lead to overloads or underutilization of resources, impacting both performance and energy consumption.

7.6.2.2 Energy Efficiency and Edge Device Performance

1. How can we minimize Edge node energy consumption while ensuring optimal performance?
Edge devices, often constrained in terms of computing power and battery life, must perform complex tasks (inference, preprocessing) without exceeding their energy budget. The goal is to optimize resource allocation (e.g., task distribution between CPU and GPU) and communication protocols to reduce consumption while maintaining a high level of service.
2. How can we integrate energy efficiency into the AI model lifecycle?
From design to maintenance, each stage of the model lifecycle (training, deployment, monitoring) has an energy impact. Automation (Python scripts, Docker/Kubernetes containers) reduces processing times and avoids redundancies but must be designed to be resource-efficient.

7.6.2.3 Reliability, Security, and Proactive Maintenance

3. How can we ensure data security in a distributed environment?
Encryption, access rights management, and protection against attacks (e.g., excessive requests) are essential, but their implementation must be consistent between cloud and Edge nodes.
4. How can we ensure proactive maintenance without overloading the infrastructure?
Performance monitoring (e.g., detecting slowdowns) and automatic model updates are crucial to prevent failures but must be lightweight and non-intrusive to avoid degrading overall performance.

Design of models distributed between Edge (periphery) and Cloud (data center) relies on balanced resource optimization to address latency, bandwidth, confidentiality, and cost constraints. Each part (Edge/Cloud) has complementary roles but also specific challenges in terms of memory, computing, security, and networking. In this deliverable, we describe how memory and computing resources used by our team are managed, as well as data security mechanisms, to optimize application performance when deploying a model on the E2CC platform.

7.6.3 Work Carried out

In this section, we describe the choices made for building the models.

7.6.3.1 Network Resource Optimization

- **Bandwidth Optimization**

Edge Method: For Armadillo, the goal is to reduce data size. We primarily aim to transfer models rather than data to the platform. Models are trained on-premise and then transferred to a central server on the platform. The estimated model size for the proof of concept can reach up to 200 MB, while use cases involving multiple models range between 400 MB and 3.3 GB.

Data Archiving and Compression Method: If data transfer remains necessary, we use a "gzip (tar)" archiving protocol followed by "Brotli" compression. This achieves a ~37% size reduction after archiving. Additionally, data is pre-selected and reduced at the Edge beforehand.

Caching Method: Available products are accessed via an API connected to our database. Queries are stored in a temporary folder for immediate reuse, avoiding redundant calls. The database structure is optimized to formalize complex queries and reduce response time.

Traffic Prioritization Method: Our database allows adding and managing rights and rules to regulate application-level traffic and prevent excessive requests.

- **Latency Reduction**

This part is primarily managed by our partners in this Work package and Work Package 1. We ensure TCP optimization at the API level, such as window size managed directly by our database.

- **Security and Reliability**

This part is primarily managed by Work Packages 5 and 6. At the application level, we ensure secure data encryption as defined by our database. The data exchange protocol uses a binary protocol protected by a Diffie-Hellman key exchange (client key: 3072 bits, server key: 3072 bits) to exchange an AES key hashed via SHA256. Subsequent exchanges use AES-GCM with KDF3 key derivation, providing a high level of security. Application access is managed via an external **Single Sign-On (SSO)** using SAML. User data storage follows an LDAP protocol, and user authentication is handled by OpenID.

7.6.3.2 Edge Processing Optimization

- Local Data Processing

Filtering and Aggregation Methods: Data are processed locally. Only models are deployed on the platform. Data preprocessing is performed at the Edge to reduce size (as defined in Deliverables 3.1 and 3.2 of Working Group 3).

Real-Time Analysis: Final models will handle text and/or image recognition, indexing, and search using AI/ML algorithms executed at the Edge, providing immediate responses without continuity.

- Edge Processing Optimization

To reduce the workload on the cloud, we chose a distributed storage solution that uses our database on the Edge nodes. This avoids remote queries.

Furthermore, our partners in this work package manage protocol prioritization for the Edge. We ensure optimization in batch data distribution during training and CPU/GPU resource allocation.

7.6.3.3 Optimized Workload for AI and Heuristic-Based Lifecycle Automation

The AI model lifecycle relies on a structured workload with key steps: **design** (algorithm development), **preparation** (data cleaning and embedding generation), **training** (parameter optimization), **evaluation** (performance validation), **deployment** (production integration via containers), and **maintenance** (continuous monitoring and updates). Each phase consumes specific resources (CPU/GPU, memory, storage) and requires precise orchestration to ensure efficiency, reproducibility, and scalability while minimizing costs and energy footprint. Automation (scripts, CI/CD pipelines) and optimization (lightweight models, Edge Computing) are essential for scaling this process. For the AI model lifecycle, we will use internal tools for algorithm development, Python scripts and GPU acceleration for automated embedding generation to ensure time savings and reproducibility, existing open-source tools for development to reduce time, test scripts for reproducibility, Docker and Kubernetes containers for easy scalability and simplified deployment and updates, and proactive monitoring with notifications for abnormal situations such as slowdowns. All these tools and methodologies will be detailed in **Deliverable 3.2**.

The solutions we have selected for optimizing Cloud-Edge infrastructures aim to design a distributed, resilient, and resource-efficient architecture. They help address challenges related to latency, bandwidth, and energy efficiency. By combining innovative methods (such as advanced compression, local processing, and intelligent orchestration) with mature tools (like Kubernetes, Docker, lightweight AI models, and databases), our work contributes to a scalable, secure, and adaptable solution for various contexts, while seeking to reduce carbon footprint and operational costs. The next phase will involve integrating these developments



into the E2CC platform and conducting thorough testing to validate performance, scalability, and energy efficiency. Collaboration with other working groups will also be key to refining and ensuring the seamless deployment of these solutions.

7.7 Ningaloo

7.7.1 Introduction

Work Package 2 aims to provide the technical building blocks for monitoring and orchestrating the platform, incorporating security and resource optimization concerns in support of energy efficiency. As Ningaloo's main work falls within Work Package 3, we are involved in Work Package 2 as early adopters of the components being developed. In this capacity, our role consists in feeding these building blocks with the information required for them to function properly, in particular the data that will enable the orchestration mechanisms to optimize the placement and execution of processing tasks.

7.7.2 Challenges

The workflows run by Ningaloo constitute a demanding use case for the Work Package 2 building blocks. They are characterized by complex sequences of interdependent processing tasks, handle a large volume of data and include a significant share of confidential data. This last dimension imposes specific requirements in terms of security, partitioning and traceability, whether for the storage, transit or execution of processing tasks. Together, these characteristics put us in a strong position to test the robustness of the Work Package 2 components under representative conditions, and to identify any gaps between the features offered and the real needs of a demanding user of the platform.

7.7.3 Work Carried out

We first contributed to the definition of the overall architecture, which made it possible to formalize the constraints that the building blocks developed in Work Package 2 must satisfy, whether in terms of interfaces, performance requirements or security requirements. In a second phase, we began a period of familiarization with and testing of the infrastructure made available; at this stage, no blocking issues have been reported. On the security side, technical discussions were held with Atos in order to frame the arrangements for integrating our workflows and handling confidential data. We have started work to set up connectors to the platform's KPI system in order to enable executions in a fully secure environment, protected against intrusions by ensuring isolation between containers. The next steps will consist in deepening this integration and broadening the scope of testing as the Work Package 2 building blocks mature.

7.8 Ryax.

7.8.1 Introduction

Ryax is an open-source, AI-driven workflow and compute management platform. It enables the execution of data analytics and AI workloads (ETL, ML, LLM pipelines) and aims at optimizing resource utilization across CPU, GPU, and RAM on heterogeneous infrastructure (cloud, on-premises, edge, HPC).

Within E2CC, Ryax serves as the orchestration runtime on top of which the project's scheduling and abstraction contributions are implemented. The platform's existing scheduler handles all safety-critical placement checks (site, architecture, GPU, and resource feasibility constraints) and provides the `first_fit` baseline. The contributions described in this deliverable extend this baseline without modifying its safety guarantees.

This deliverable covers the first months of Ryax's contributions to Work Package 2, across Tasks 2.4 and 2.5. It presents the architectural foundations and initial implementations enabling multi-infrastructure workflow orchestration with multi-criteria placement

7.8.2 Challenges

Task 2.4 - Intermittency and federated resource management. Edge clusters in the continuum may disconnect from the central control plane for extended periods, due to mobility, network failure, or planned isolation. Orchestration must remain locally operational during these periods and resynchronize cleanly upon reconnection, without data loss or duplication.

Task 2.5 - Application development and deployment. Applications targeting the edge-cloud continuum must be composable, annotatable, and deployable across heterogeneous infrastructure (varying architectures, tiers, and hardware constraints) without requiring deep expertise in distributed systems.

In both cases, a transverse challenge is multi-criteria placement: among feasible candidates for a given task, the choice is not neutral. One cluster may be cheaper, another faster.

7.8.3 Work Carried out

7.8.3.1 Task 2.4 - Multi-criteria placement policy

We consider an architecture where each edge cluster is autonomous, running its own local Kubernetes scheduler. The federation layer adds three capabilities on top of this baseline: online data exchange between clusters, offline buffered local execution, and state resynchronization upon reconnection.

The core contribution is an opt-in placement policy for the Ryax runner. It operates exclusively on the set of candidates that have already passed all safety filters, computes a deterministic ranking, and returns the ordered list to the normal `first_fit` placement path.

For each feasible candidate, the runner computes two lower-is-better criteria:

Criterion	Definition
Predicted runtime	Expected execution time on the candidate (seconds)
Estimated run cost	$\text{hourly_cost} \times \text{predicted_runtime} / 3600$

TABLE 1 : CRITERIA

Candidates are ranked by their geometric distance from an ideal point (lowest runtime, lowest cost) and an anti-ideal point (highest runtime, highest cost). The method is deterministic, handles criteria of different scales, and introduces no hidden defaults.

The performance/cost trade-off is exposed via the action's existing objective weights, combined into a single scalar slider

$$\text{slider} = \text{performance_weight} / (\text{performance_weight} + \text{cost_weight})$$

The slider produces a continuous scale between two extremes: a value of 1.0 gives full priority to minimizing runtime regardless of cost, while a value of 0.0 gives full priority to minimizing cost regardless of runtime. Intermediate values produce a weighted trade-off, with 0.5 treating both criteria equally. In practice, the slider is the first configuration choice a user makes: it determines how the ranking is computed before any candidate is evaluated. Missing, invalid, or unsupported weights (including a non-zero energy weight, reserved for a future version) invalidate the ranking and trigger a fallback to the safety-critical placement, with the reason recorded.

7.8.3.2 Task 2.5 - Application development and deployment abstractions

Applications are expressed as **modular DAG workflows** composed of typed, independent components. The abstraction layer provides:

1. **Zero-code composition** via a declarative YAML/JSON format and a graphical designer;
2. **Unified batch and stream processing**: a single workflow definition supports both modes, selected at deployment time based on the data source;
3. **Resource and constraint annotations**: each component declares hardware needs (`requires: GPU`), tier preferences (`tier: edge`), and resource budgets (CPU, RAM, bandwidth).

These annotations feed directly into the runner pipeline: hardware annotations are consumed by the feasibility filters, resource budgets by the resource feasibility check, and objective weights by the ranking slider.

Each component is compiled into a container image or serverless function using **Nix**, producing reproducible hermetically sealed builds, multi-architecture images (x86_64 and ARM64) from a single component definition, and minimal runtime environments with a reduced attack surface.

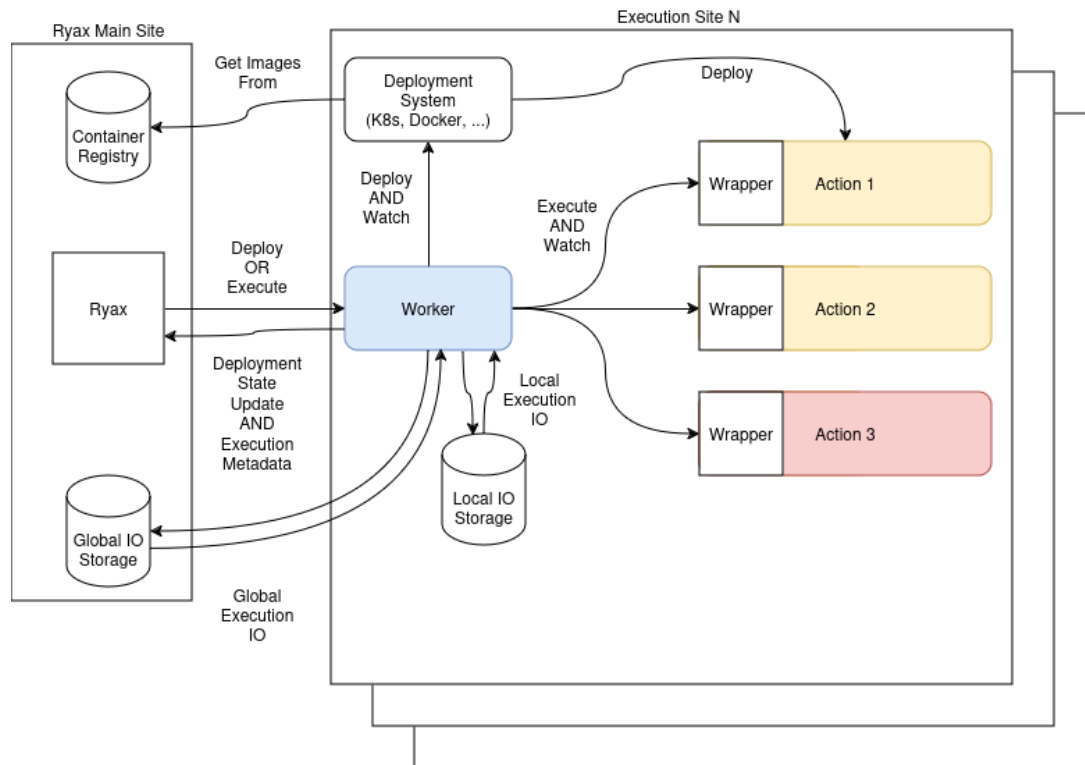


FIGURE 13 : ARCHITECTURE RYAX – SITE LEVEL VIEW.

The figure illustrates the role of the *Worker* component in the Ryax architecture. The *Worker* acts as the bridge between the Ryax platform and a local compute site, handling the full execution lifecycle: resource allocation, action deployment, data fetching, execution triggering, and result reporting. Each *Worker* registers itself independently with the platform and manages its own local storage, operating autonomously from other sites.

This design enables native multi-infrastructure support: each *Worker* instance is associated with a distinct infrastructure, whether a cloud kubernetes, a bridge to an on-premises cluster, or an edge site, and operates independently. Workflows can therefore span multiple *Workers* simultaneously, with data exchanged between sites through a shared filestore when an execution crosses infrastructure boundaries. This architecture is the foundation on which Ryax's multi-infrastructure orchestration capabilities are built within E2CC.

The immediate next step is the extension of the placement policy to incorporate energy consumption and carbon emission objectives, currently deferred as a fallback when a non-zero energy weight is provided. This extension depends on the availability of energy and carbon metrics from the monitoring infrastructure developed in WP4. Dynamic reconfiguration (updating component placement at runtime without restarting the workflow) is also planned. A broader evaluation across real-world workloads will be conducted to validate the scheduling decisions across the full set of objectives.

7.9 NBS

7.9.1 Introduction

Running containers under the standard runtime means every workload shares the host kernel. For a lot of workloads that's fine, but it isn't always good enough.

Untrusted or third-party code, tenants sharing the same cluster, or workloads with stricter security and compliance often need stronger boundaries. gVisor, Kata Containers, and KubeVirt each aims to provide that in different ways, but getting them onto a cluster normally means setting each one up by hand on every node and keeping it working over time or specializing node pools for each or even worst specializing full kubernetes clusters for each.

The runtime-operator is a Kubernetes operator for installing and managing alternative container runtimes — gVisor, Kata Containers, and KubeVirt on bare-metal clusters. It lets you describe what you want with a custom resource called RuntimeConfig (which runtime, which nodes, which version), and the operator does the rest: it installs the runtime on the nodes, sets up the matching RuntimeClass, tracks the installation per node, handles version upgrades, and cleans everything up when the config is deleted.

It treats runtime installation and lifecycle as something you declare through Kubernetes resources, and it manages the work across the cluster the same way Kubernetes manages everything else. The result is that a platform team can offer stronger isolation as a normal cluster feature, without maintaining their own per-node tooling.

7.9.2 Challenges

Kubernetes only runs workloads under one default runtime by default, and it has no built-in way to install or manage other runtimes across all nodes of the cluster. For workloads that need stronger isolation, need to run untrusted code, use multitenancy, or things that really need a full VM, there's no proper, declarative way to set that up on the platform.

This project exists to fill that gap, by managing these runtimes through normal Kubernetes API objects rather than manual work on each node.

The tasks/actions it aims to let you do that you can't do out of the box:

- Query for gVisor, Kata, or KubeVirt through a single resource, instead of setting each runtime up by hand on every node.
- Installed and configure the required runtime on the chosen nodes automatically, with the status of each node visible as it happens.
- Create, update, and remove the related RuntimeClass with the install, so workloads can use the runtime straight away.
- Roll out a new runtime version across the fleet in a controlled way (rolling, blue-green, canary, or drain-and-replace).
- Check that runtime images are what they claim to be before they're installed, and pin what gets deployed to the exact verified image.
- See the state of the runtimes through ordinary Prometheus and Grafana, via metrics and alerts the operator exposes.

The global goal is to offer one consistent way to run stronger isolation on Kubernetes using resources dev,devops,devsecops teams already understand, rather than a pile of shell scripts or IaC modules.

7.9.3 Work Carried out

The work done so far covers not only the operator code (the controllers, the reconciliation logic, the update strategies, the API, ...), but also the runtime packaging in OCI images and their related installer scripts and tools for each of the selected runtime (gVisor, Kata, and KubeVirt), the automation that builds and signs those images, the observability (metrics and alerting primarily), and the documentation. It came to 58 items split across six stages. Each item went the same way: a failing test first, then the smallest change that makes it pass, then the local checks, then a review, then a commit on its own. Each stage finished with a validation pass and a sign-off before the next one started.

- Automation checks. The checks had been set up so they could never fail. They now block bad changes properly by linting, testing, and security scanning.
- Installation. The gVisor, Kata, and KubeVirt install process has been implemented and tested as a first step; the Job-based installer in addition to the DaemonSet installer is also a work in progress.
- Upgrades. The four update strategies have been implemented and need more testing in a real cluster to report success or failure. They move node runtime installation to the target version and keep track of their progress across restarts.
- Availability and security. The controller-runtime and Kubernetes libraries were designed and implemented. The health check probes now look at the operator's own health, so it cannot get stuck on itself. Builds for multiple architectures have not been tested yet.
- Observability. The monitoring configuration now ships by default. Alerts were added for the operator going down and for installations getting stuck or failing, which cover a readiness check that was removed earlier. An end-to-end test should be implemented for the four update strategies.
- Documentation. A very early documentation has been produced but not reviewed or improved yet.

What's left (as global topics): the operator still needs to be run against a real cluster before any real use.

Kata and KubeVirt both failed when last tested on a virtual cluster and need to be fixed and retested.

Canary and drain-and-replace rollbacks do not yet undo the version change on each node — only rolling and blue-green do.

Managing runtimes via jobs rather than DaemonSets is also something that needs to be explored and improved.

8 CONCLUSION AND NEXT STEPS

This step of the project confirms the relevance of the technical orientations adopted for the orchestration of the Cloud-Edge continuum. The work carried out by the partners has made it possible to consolidate the main foundations of the orchestration layer, covering multi-cluster management, crisis-aware scheduling, workload migration, runtime isolation, workflow placement, monitoring, security integration, and resource optimization. These contributions provide a coherent basis for building a secure, interoperable, and energy-aware orchestration framework adapted to heterogeneous infrastructures.

The next steps will focus on the integration and validation of these components within the E2CC testbed. This includes finalizing the implementation of the selected orchestration mechanisms, strengthening interoperability between the different building blocks, and validating their behavior under representative Cloud-Edge scenarios. Particular attention will be given to the integration of security mechanisms, observability tools, energy and carbon metrics, and workload placement strategies in order to ensure that orchestration decisions can effectively balance performance, resilience, cost, and sustainability objectives.

Particular attention should also be paid to the potential risks of overlap or collision between the different orchestration-related developments carried out by the partners. As several components may address similar functions or interact within the same orchestration layer, the next phase should include a consolidation effort to clarify component boundaries, identify complementarities, and ensure coherent integration within the E2CC framework.

In the upcoming phase, partners will also continue experimental evaluations on real or representative infrastructures, address the remaining technical limitations identified during the first developments, and refine the interfaces required for cross-work-package integration. These activities will support the progressive maturation of the E2CC orchestration framework and prepare the transition from initial prototypes toward a more complete, operational, and validated solution for secure and sustainable Cloud-Edge orchestration.

9 ACRONYMES ET ABREVIATIONS

Term	Definition
AI	Artificial Intelligence
API	Application Programming Interface
E2CC	Eco Edge to Cloud Computing
IAM	Identity and Access Management
IaaS	Infrastructure as a Service
ML	Machine Learning
SLA	Service Level Agreement
TLS	Transport Layer Security

TABLE 2 : ACRONYMS AND ABBREVIATIONS