Deliverable D4.2: Best Practices to Reduce Data Transfers Across Data Centers

Deliverable Name	Best Practices to Reduce Data Transfers Across Data Centers
Deliverable Number	D4.2
Deliverable Related Number	D4
Workpackage	WP4
Deliverable Leader	FORTH
Date of Delivery	11/08/2026
Deliverable Type	Other
Dissemination level	PU

Peer reviewer Name	**Organization**
Giannis Petsis	FORTH
Dimitris Xenakis	CERN

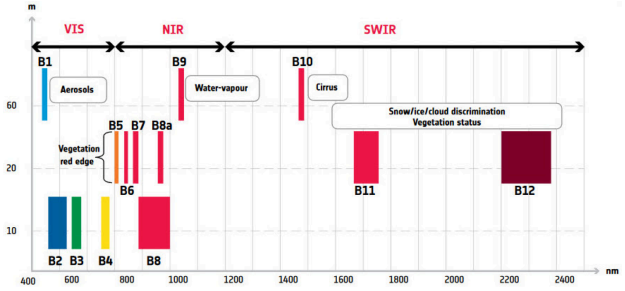
Version	**Author**	**Date**	**Comment**
1.0	Giorgos Saloustros	8/7/2026	First version
1.1	Giorgos Saloustros	22/7/2026	Address review comments
1.2	Antony Chazapis	22/7/2026	Internal review

1.3	Antonis Tapanlis	23/7/2026	Multisite architecture section
1.4	Giannis Petsis	23/7/2026	HPK-2.0 improvements
1.5	Giorgos Saloustros	28/7/2026	Address review comments by D. Xenakis

Table of Contents

1. Terminology

- **AI: A**rtificial **I**ntelligence.
- **Copernicus:** the Earth Observation component of the European Union's space program, looking at our planet and its environment for the benefit of Europe's citizens.
- **CPU: C**entral **P**rocessing **U**nit; the general processing unit without constraint on the instruction set (e.g., x86, ARM).
- **DASI: D**ata **A**ccess and **S**torage **I**nterface; a metadata-driven data store developed by ECMWF that acts as a semantic interface for data, where the data is indexed and uniquely identified by sets of scientifically-meaningful metadata keys.
- **DL: D**eep **L**earning; the subset of machine learning methods based on ANNs.
- **EO: E**arth **O**bservation.
- **External metadata:** metadata which refers to the object embedding data (e.g., file size).
- **GIS: G**eographic **I**nformation **S**ystem; a computer-based tool that analyzes, stores, manipulates, and visualizes geographic information, usually in the form of maps.
- **GPU: G**raphical **P**rocessing **U**nit; a specialized processor originally designed to accelerate computer graphics rendering, featuring a highly parallel architecture that excels at data-parallel operations (nowadays, it serves as the primary computational engine for AI workloads).
- **Helm:** a package manager for Kubernetes that simplifies deploying and managing applications by packaging them into reusable charts.
- **Hierarchical schema:** a structured framework organizing metadata into a layered arrangement, mirroring the structure of ontologies. This approach groups information by categories—such as internal and external metadata—allowing for more systematic classification and retrieval.
- **HPC: H**igh **P**erformance **C**omputing.
- **Image segmentation:** a process of partitioning a digital image into multiple image segments, also known as image regions or image objects (sets of pixels).
- **Internal metadata**, or **Semantic metadata:** metadata which refers to the content of the data (e.g., the maximum value of parameters fields which the meteorological data file).
- **Kubernetes namespace:** a logical partition in a cluster, used to organize and isolate resources for different teams, projects, or environments.
- **Machine-to-Machine:** internal interactions between two components within the DaFab infrastructure—such as a metadata generator pushing data to the Unified Metadata—operating entirely within the system's domain.
- **Metadata catalog:** a database that allows querying and holds descriptive information about data objects (e.g., coordinates associated with a satellite image stored in a remote datahub).
- **ML: M**achine **L**earning; a branch of artificial intelligence focused on creating and applying statistical algorithms that learn from data and generalize to unseen data, and thus perform tasks without explicit instructions.
- **MSI: M**ulti-**S**pectral **I**nstrument; a sensor onboard the Sentinel-2 satellites that collects imagery across multiple spectral bands, each with its own spatial resolution



(the figure shows basic band characteristics
including their resolution in Y axis).

- **OAuth: O**pen **Auth**orization; a protocol that lets users grant apps secure access to their data on another service—without sharing login credentials—by exchanging tokens instead of passwords.
- **RUCIO:** a data management system developed at CERN, designed to handle large volumes of data across various scientific collaborations.
- **Semantic Web:** a World Wide Web extension (historically referred to as Web3) that encodes data with explicit semantics to describe ontologies, relationships, and categories of entities, thereby enabling advanced data integration, reasoning, and interaction across heterogeneous sources.
- **Sentinel-2:** a Copernicus mission comprising two identical satellites in the same orbit, each equipped with an innovative wide-swath, high-resolution multispectral imager featuring 13 spectral bands to provide new perspectives on land and vegetation.
- **Sentinel2 L0 level:** reconstructed raw MSI measurements with minimal ancillary info needed for further processing (not distributed to end users).
- **Sentinel2 L1A level:** uncompressed raw MSI measurements with coarsely co-registered spectral bands and ancillary info such as geometric/radiometric calibrations (not distributed to end users).
- **Sentinel2 L1B level:** radiometrically corrected multi-spectral MSI images, featuring coarsely co-registered spectral bands but lacking the refined geometric corrections or orthorectification.
- **Sentinel2 L1C level:** ortho-rectified, UTM-geocoded top-of-atmosphere reflectance images as 100km x 100km tiles, with sub-pixel multispectral / multi-date registration (accessible to all users).
- **Sentinel2 L2A level:** data derived from L1C imagery through atmospheric correction, providing ortho-rectified, UTM-geocoded bottom-of-atmosphere reflectance along with Aerosol Optical Thickness, Water Vapour, Scene Classification, and Quality Indicators (accessible to all users).
- **STAC: S**patio**T**emporal **A**sset **C**atalog; an open specification defining a standardized, JSON-based structure to describe geospatial and temporal metadata, enabling easier search, discovery, and access to a wide range of remote sensing and related datasets.
- **ToI**: Time-of-Interest
- **User:** an external actor interacting with the DaFab system—a distributed framework of interacting components capable of interfacing with the outside world—primarily by submitting queries through the Unified Metadata Catalog to obtain dataset descriptions (with Machine-to-Machine interactions considered internal users).

2. Executive Summary

Deliverable D4.2 presents the architectural design of the data-aware federated workflow infrastructure developed within Task 4.2 of the DaFab project. The primary objective is to reduce data transfers across geographically distributed, heterogeneous cloud and High-Performance Computing (HPC) infrastructures in Europe.

We start from the observation that Copernicus Earth Observation (EO) workflows which process satellite imagery products have two basic characteristics. First, there is massive inherent parallelism: scientists must run the same analysis for thousands of individual Copernicus products simultaneously, with no data dependencies between them. Second, these workflows are data-intensive; our measurements in the project's pilot use cases show that intermediate execution artifacts can expand up to 12 times the size of the original input. Moving intermediate data between different data centers or administrative domains creates severe network bottlenecks.

To handle this complexity, our strategy is to make use of the independence of these thousands of tasks by intelligently placing them where the Copernicus original products already sits. By treating the entire European infrastructure as a single, data-aware pool, we can avoid unnecessary movement except in specific cases where specialized hardware, such as GPUs, is required for a subset of the workflow and it is in a different site. To run every workflow where the data is, the architecture ensures full portability, by enabling workflows to execute across any environment without being locked into a specific site's local storage stack or administrative constraints.

The DaFab architecture addresses the challenge of data gravity through two integrated pillars:

- ***Pillar 1: A Unified, multi-site computational substrate.*** We break down the silos between Cloud-native and HPC environments by unifiying them in a federation through Karmada and HPK-2.0. HPK-2.0 provides a user-level networking stack that allows Kubernetes workloads to run on Slurm-managed HPC facilities without administrator privileges. By registering these HPK-managed HPC islands as member clusters within a global Karmada federation, DaFab creates a "run anywhere" environment. This allows the system to transparently schedule EO workflow steps across Europe's most powerful CPU and GPU resources as if they were a single, elastic cloud.

- ***Pillar 2: Data-Aware Orchestration and Artifact-Native Dependencies.*** To eliminate the reliance on specific storage stacks which locks workflows to a subset of sites we transition to an artifact-native model using Argo Workflows. In this model, data dependencies are resolved via a global discovery layer where Rucio acts as the authoritative index for artifact locations. This enables cooperative caching and locality-aware scheduling: To schedule a workflow, the scheduler queries Rucio to locate the required input (artifact) id. The system then applies "cost-aware" logic to either move the computation (the entire workflow) to the data (reducing transfer) or,

in the case of specialized hardware needs (e.g., GPUs), to perform a calculated "site-jump" for that sub-graph of the workflow.

Together, these mechanisms ensure that massive intermediate data (often 12x the original input) remains local to the compute site whenever possible. By bridging the Cloud-HPC divide and decoupling data from the underlying storage technology, DaFab provides the best practices and technical foundations for the efficient, scalable, and data-responsible processing of Copernicus EO products across the European federation.

3. Introduction

The DaFab project enables Earth Observation (EO) workflows to operate efficiently across a geographically distributed federation of cloud and High-Performance Computing (HPC) infrastructures in Europe. In this landscape, the volume of Copernicus data is substantial, but the movement of this data is further complicated by the diverse nature of European infrastructure. Participating sites are separated by distinct administrative boundaries and utilize vastly different technological stacks:

- Cloud-Native Sites: These typically utilize Kubernetes for orchestration and Object Storage (e.g., S3-compatible) for data handling, offering high flexibility for microservice-oriented architectures.
- HPC Facilities: Europe has made massive investments in large-scale supercomputing centers via the EuroHPC Joint Undertaking, funding world-class systems such as Vega (Slovenia) and MeluXina (Luxembourg). These sites operate under traditional Batch Management systems (e.g., Slurm) and rely on high-performance parallel filesystems like Lustre or GPFS.

A central challenge for DaFab is to bridge these two worlds. While the EuroHPC sites provide the massive GPU and CPU power necessary for large-scale AI and complex simulations, they often exist as "walled gardens" with storage technologies that are not natively compatible with cloud-native EO workflows. The administrative complexity of moving data from a cloud-based Copernicus repository to a Slurm-based HPC cluster can create significant bottlenecks, preventing the European EO ecosystem from making effective use of these large-scale investments.

To reduce data transfers in the EO domain, we must first consider the specific structure of typical workloads, such as field delineation or water analysis studied in the context of the DaFab project. These workflows are executed per Copernicus [15] product and follow a distinct serial pattern: a CPU-intensive pre-processing stage, an AI-driven inference stage requiring specialized GPU resources, and a final CPU-heavy post-processing stage. These three critical characteristics of these workflows dictate our optimization strategy:

- Data Expansion: EO workflows produce significant volumes of intermediate data often up to twelve times the size of the original Copernicus input. Moving these intermediate results across network boundaries is often more costly than the computation itself.
- Heterogeneous Resource Demands: While pre- and post-processing can run on commodity cloud CPUs, the AI core requires specialized accelerators often found only in specific HPC centers.
- Administrative and Technical Silos: The "big machines" in Europe often operate under Slurm-based HPC environments, while smaller providers are cloud-native. This creates a friction point for any workflow that needs to span across different administrative domains.

Based on these domain-specific observations, we design the DaFab architecture around three lessons learned that map directly to our technical implementation:

1. ***Location-Aware Parallelism and Affinity:*** To increase federation-wide utilization, the system should schedule each complete per-product workflow at a single eligible site whenever possible, ensuring that expanded intermediate data remains local throughout processing. The primary rule is to honor data gravity: if a Copernicus product already resides at an eligible site, the workflow should run there.

   A subgraph site jump may be considered only as a cost-model-driven exception, typically for GPU-heavy AI stages when the local site lacks the required accelerators. Such a jump should be made only where the expected accelerator benefit outweighs transfer time, queue time, packaging overhead, and the cost of moving intermediate data. This approach reduces unnecessary movement of the 12x expanded intermediates by keeping CPU-bound stages colocated with storage.

2. ***Universal Portability through HPK-2.0 and Federation:*** A product workflow must be able to execute on any site, regardless of whether it is a cloud environment or a Slurm-managed HPC cluster. DaFab achieves this through HPK-2.0 (Section 4), which enables Kubernetes-oriented workloads to execute on HPC facilities without administrator privileges. By layering a multi-cluster federation (Section 5) over these diverse sites, we treat the entire European infrastructure as a single, cloud-native substrate.
3. ***Artifact-Native Workflows::*** To prevent workflows from being locked to a specific site's storage technology (e.g., Lustre vs. S3), we have decoupled data handling from the application logic. By transitioning to Argo artifact-based definitions (Section 5), users no longer hard-code file paths. The orchestration layer manages the inputs and outputs, allowing steps to be placed dynamically based on resource availability rather than storage assumptions.
4. ***Storage-Agnostic Discovery:*** We utilize Rucio as a global metadata and discovery service for Copernicus products. Crucially, Rucio remains independent of the local storage stack, providing a unified view of input datasets, cached replicas, and generated artifacts across all administrative domains.

These capabilities enable a cost-aware scheduling policy that determines whether to execute a step where the data resides or to stage sub-parts of the workflow to a site with specialized accelerators. In all cases, data movement becomes an explicit, controlled, and cost-model-driven decision.

The remainder of this deliverable describes the technical realization of these pillars: the HPK-2.0 architecture for HPC integration, the multi-site federation logic using Karmada, the implementation of Argo workflows over a multisite federation, and the practical transition of EO workflows to a multisite, artifact-native model. These mechanisms support the scalable, efficient, and data-aware execution of distributed Copernicus workflows while reducing unnecessary data movement across the federation. In Figure 3.1, we show the system's architecture.

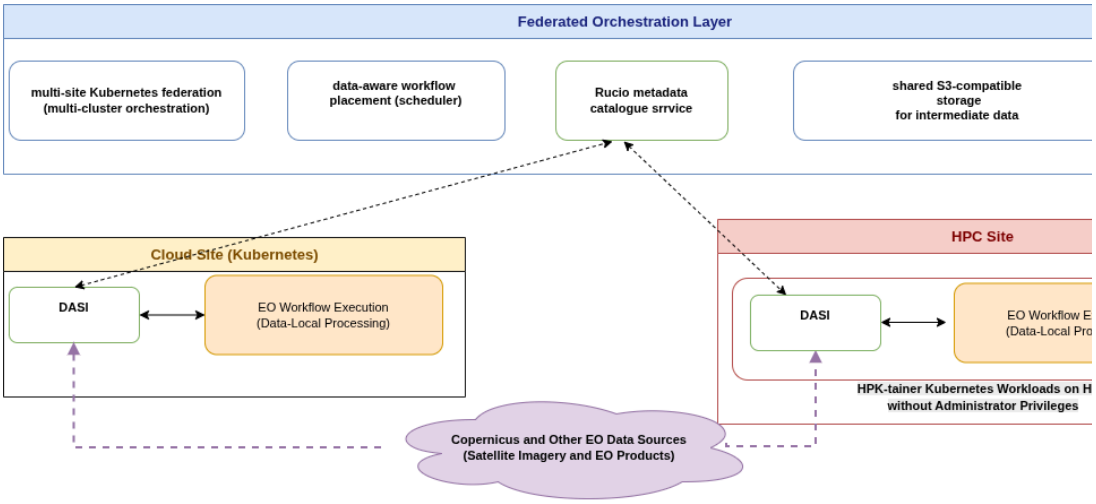


Figure 3.1: System Overview

4. HPK-2.0

Running Kubernetes (K8S) on top of a Slurm-managed HPC system requires (a) a translation layer between K8S objects and Slurm resources and (b) a private, routable network that connects containers deployed across multiple allocated compute nodes. The second capability is mandatory for complex Earth Observation (EO) pipelines, particularly those utilizing Argo-based workflows. In these scenarios, a private network is essential for orchestrating multi-stage data processing tasks. For example, the Argo controller must communicate with transient worker pods to manage execution states and coordinate the processing of massive spatial datasets across a distributed environment.

As described in Deliverable D4.1, the original HPK design addresses both (a) and (b) requirements. However, for (b), it relies on close integration with the underlying HPC host infrastructure to provide private networking for K8S. More precisely, HPK uses Flannel [7] and its supporting software ecosystem, which must be installed and managed directly at the host level. Flannel requires root privileges because it must perform low-level system operations that are restricted to administrators. These include creating virtual network interfaces (like vxlan or bridge), modifying the host's kernel routing tables to direct traffic between nodes, and manipulating the system firewall (iptables) to manage packet forwarding and security.

In most HPC environments (Meluxina, Vega), these elevated permissions are not available to general users to prevent them from interfering with the shared network configuration of the cluster. Consequently, the deployment of HPK depends entirely on system administrators'

intervention, as an ordinary user cannot modify the networking stack of a compute node once it has been allocated

To overcome these limitations, we design ***HPK-2.0.*** HPK-2.0 is a user-level networking framework that enables cloud-native Kubernetes workloads to run in High-Performance Computing (HPC) facilities where users do not have administrator privileges. HPK-2.0's architecture addresses the root challenge in High-Performance Computing (HPC) systems by implementing the entire private network stack of K8S outside the host operating system and into the user space. As a result, the software runs with the user's own limited permissions. HPK-2.0 achieves this through a nested container design: it places a workload container inside a user network namespace. This separation creates a private environment that is entirely decoupled from the host's networking configuration, removing the need for system-wide installations or administrative (root) privileges.

4.1 HPK-2.0 Overview

To provide a clear understanding of the HPK-2.0 architecture, we describe it in a top-down fashion: beginning with the high-level cluster allocation, moving into the node-level "bubble" environments, and finally detailing the inner-container workload execution.

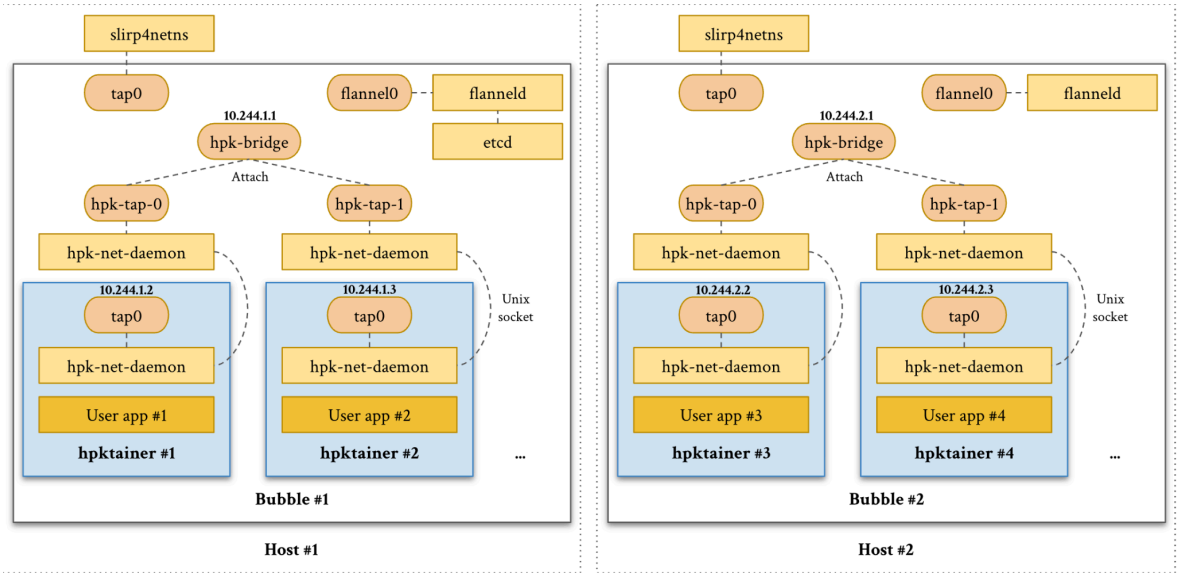


Figure 4.1: HPK-2.0 architecture

At its core, HPK-2.0 moves the Kubernetes overlay network from the host operating system into user space by utilizing a nested architecture of rootless containers—essentially running one container inside another. Both layers are built using Apptainer [9], a container platform designed to package applications and their dependencies into portable, isolated environments optimized for HPC systems. By operating entirely within a standard Slurm allocation, HPK-2.0 remains strictly isolated from the host's networking configuration.

This architecture is structured across three distinct operational tiers:

1. **Cluster Allocation:** The framework operates entirely within a standard batch job allocation provided by Slurm. To the HPC system administrators, HPK-2.0 appears as a single, multi-node user application running under standard, unprivileged permissions.
2. **The Outer Layer (Node-Level Bubbles):** On each allocated compute node, HPK-2.0 launches an outer container (the "bubble"). This layer acts as an isolated networking environment, abstracting the host network and running all the necessary user-space routing daemons and overlay interfaces (Flannel or Calico).
3. **The Inner Layer (Workload Execution):** Inside these bubbles, HPK-2.0 launches the nested containers that execute the actual K8S workloads. These inner containers operate as if they are part of a standard cloud-native environment, seamlessly communicating over the overlay network established by the surrounding bubbles.

This separation of concerns allows users to deploy and manage complex, multi-node cloud-native applications independently, without requiring system-wide installations, host-level kernel modifications, or root privileges.

4.2 Node-Level Bubbles

The outer layer, referred to as a "bubble," is deployed on each allocated HPC node and hosts the networking components required for the cross-node overlay. Bubbles provide unprivileged connectivity by creating a virtual gateway and translating data traffic in user space. This allows the bubble to securely send and receive data across external networks without modifying the host's core operating system settings or requiring administrative permissions.

To enable this connectivity in a fully unprivileged HPC environment, HPK-tainer integrates a multi-layered user-space networking stack inside the bubble. This stack combines kernel-provided virtual network interfaces, user-space packet processing, and an overlay network for multi-node communication. The main networking components encapsulated within the bubble are detailed below.

4.2.1 TAP interface

The connection between the container network namespace and the user-space networking layer is provided through a TAP interface. A TAP interface is a Linux kernel virtual network device operating at Layer 2 of the network stack, meaning that it transports complete Ethernet frames rather than only IP packets.

Unlike a physical network interface card (NIC), a TAP interface has no physical cable or hardware port. Instead, it provides a virtual Ethernet endpoint. From the container's perspective, it behaves like a conventional network interface: the container can assign it an IP address, configure routes, and send or receive network traffic through it. On the other side, the interface is exposed to a user-space process through a file descriptor. This allows a user-space networking component to read Ethernet frames sent by the container and write return traffic back to the container.

In the HPK-2.0 architecture, the TAP interface therefore provides the handoff point between the Linux kernel networking stack in the container namespace and the user-space networking process (slirp4netns). It allows applications to use standard network interfaces and protocols without requiring direct access to privileged host networking resources.

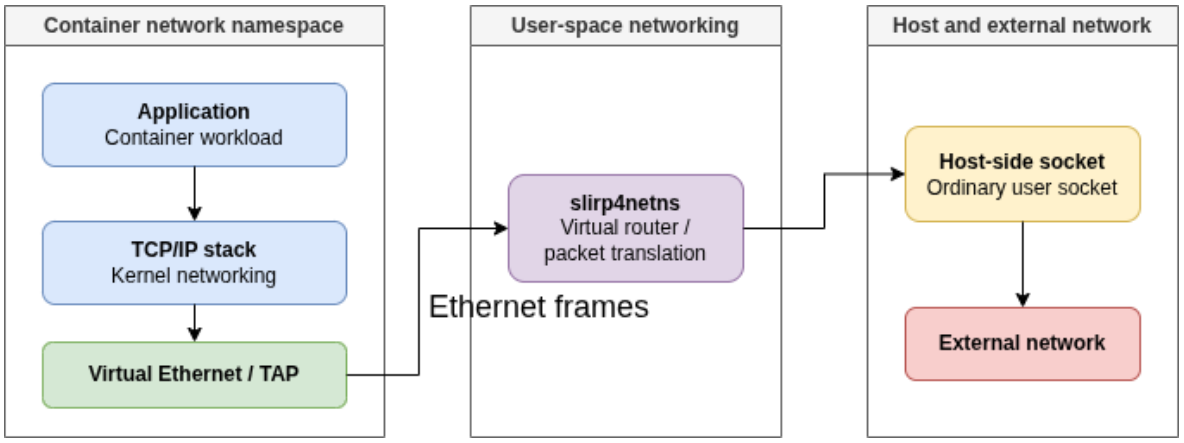


Figure 4.2: Container to external network data path through the TAP interface

A simplified outbound data path, as shown in Figure 4.2, works as follows:
An application inside a container sends data through its local TCP/IP stack to a virtual Ethernet or TAP interface. The slirp4netns process then translates this traffic in user space, passing it through a host-side socket to reach the external network.

For return traffic, the path is followed in reverse: slirp4netns receives data through a host-side socket, constructs the corresponding network packets, and writes them to the TAP interface for delivery to the application in the container.

4.2.2 slirp4netns

slirp4netns [12] provides the core rootless networking capability of HPK-tainer. Conventional container networking typically relies on privileged host operations, including creating network bridges, virtual Ethernet pairs, routing rules, and Network Address Translation (NAT) rules. Such operations are generally unavailable to ordinary HPC users.

slirp4netns addresses this restriction by functioning as a user-space virtual router. It connects to the TAP interface associated with the container network namespace and processes network traffic without modifying the host's network configuration. It uses a user-space network stack to translate container traffic into ordinary host-side network connections, which are created with the permissions of the user running the workload.

This approach gives the container an isolated network environment and private addressing while avoiding direct privileged access to the host network stack. The container can use a standard, unmodified TCP/IP stack and sees a regular Ethernet-style interface, such as

tap0. At the same time, the host only observes normal user-owned processes and sockets rather than privileged network configuration changes.

4.2.3 High-Speed Forwarding Service

The ***hpk-net-daemon*** is a specialized Python service developed for the HPK-tainer project. It coordinates and forwards network data between the nested container networking layers and the host-side networking components.

The daemon uses UNIX-domain sockets for local inter-process communication. Since UNIX-domain sockets operate entirely within a single host and do not require traffic to traverse an external network interface, they provide an efficient and low-latency mechanism for exchanging data between the relevant user-space processes.

By using this local forwarding mechanism, hpk-net-daemon reduces coordination overhead between the networking layers and supports efficient data transfer within the HPK-tainer architecture.

4.2.4 Container Network Interface (CNI)

A central architectural choice in HPK-tainer is the Container Network Interface (CNI), which assigns private IP addresses to containers and determines how traffic is routed both within a bubble and across bubbles on different HPC nodes. HPK-tainer supports two CNI configurations to accommodate different HPC restrictions: Flannel and Calico. Flannel acts as a lightweight provider that creates a logical Layer-2 overlay network via a VXLAN interface, allowing separated workloads to communicate as if on the same local network. However, Flannel relies on Linux bridge networking, which requires kernel modules (like vxlan and br_netfilter) that unprivileged users cannot always dynamically load. To address this, HPK-tainer also supports Calico, a Layer-3 CNI that avoids the Linux bridge model entirely. Calico distributes routing information by establishing a BGP-based routing mesh through a single forwarded host-level TCP port (17900) and uses proxy ARP for workload connectivity inside the bubble. By supporting both, HPK-tainer provides the flexibility to deploy a routable private overlay network regardless of whether specific kernel modules are available to the user.

4.2.5 Data Path Operations

Within each bubble, HPK-tainer launches nested Apptainer containers that represent the user workloads (inner layer). These containers are attached to the bubble's local subnet through TAP interfaces, a Linux bridge, and the custom hpk-net-daemon. The forwarding mechanism connects interfaces through UNIX-domain sockets (shared between namespaces), allowing packets to be transferred between the nested container and the overlay network without requiring privileged networking services on the HPC host. Each workload container receives a routable private IP address, invisible from the host, and can therefore communicate with containers hosted in bubbles on other allocated nodes.

The following scenarios illustrate how the CNI daemon, etcd, and user applications reside within their own containerized namespaces, bridging networking traffic entirely within the container environment depending on the destination.

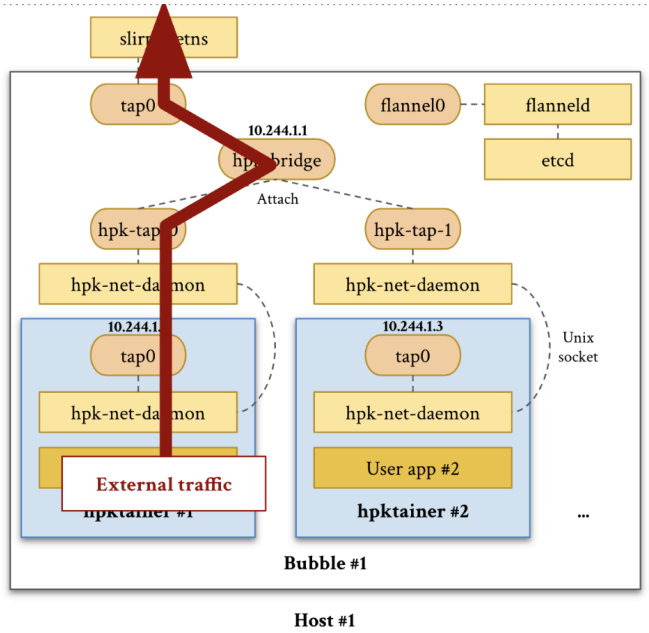


Figure 4.3: Example of external traffic

- **Outbound External Traffic:** As shown in Figure 4.3, when an application requires access to external networks, data routes through the virtual interface to the hpk-net-daemon. It is forwarded to the hpk-bridge inside the bubble, and exits the host through the tap0 interface provided by slirp4netns.

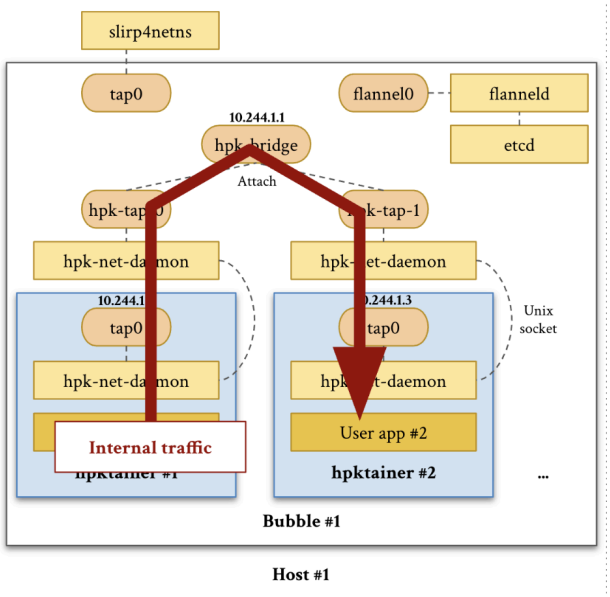


Figure 4.4: Example of internal traffic

- **Intra-node Traffic:** As shown in Figure 4.4, for communication between two containers on the same physical host (same bubble), data travels from the source container to the hpk-bridge. Since both containers are attached to this common bridge, traffic is simply routed down to the target application.

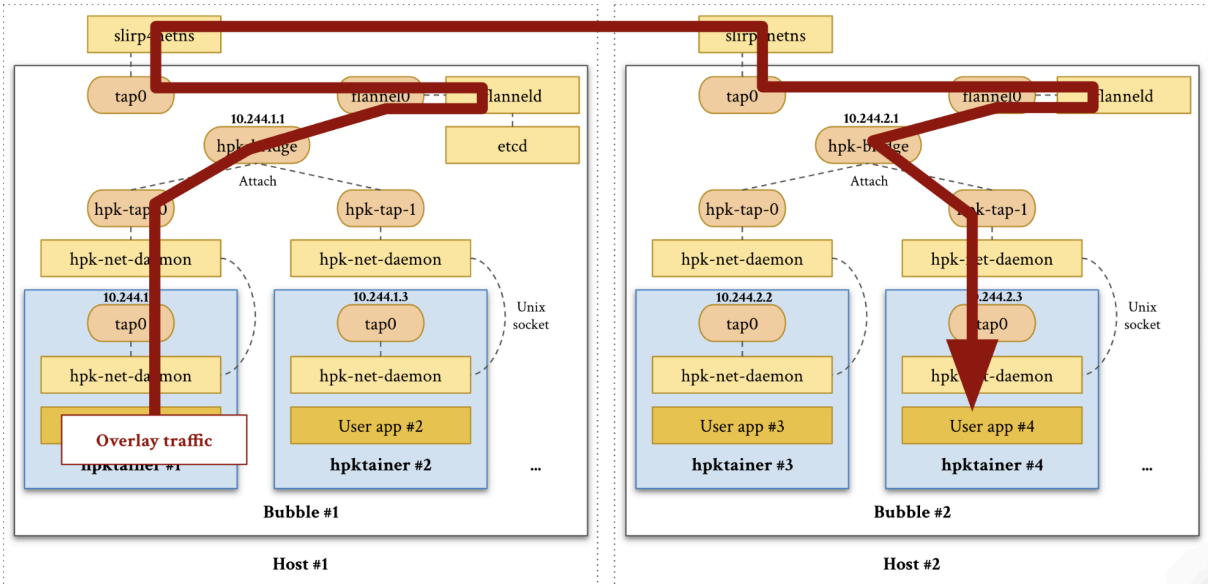


Figure 4.5: Example of overlay traffic

- **Inter-node Traffic:** As illustrated in Figure 4.5, when a container on one host communicates with a container on a different host, traffic travels to the local hpk-bridge and is directed to the flannel0 interface, provided by the Flannel CNI daemon. The CNI encapsulates the packets and routes them outward via slirp4netns, traveling across the physical HPC network to the destination node, where it follows the reverse path.

4.3 Kubernetes Integration

HPK-tainer shifts the execution environment of standard Kubernetes and orchestration services so that they function securely within an unprivileged batch-scheduled architecture.

To coordinate the cluster, one specific bubble acts as a controller, hosting services like k3s (a lightweight Kubernetes binary) [11] and etcd (K8S distributed data store) [10]**.** The operational lifecycle for the majority of these supporting services remains unchanged from the previous HPK model. The primary evolution is simply the shift in the execution environment—these services are now encapsulated within the bubble rather than being run directly on the host, gaining the portability and isolation benefits of the new architecture.

Both the original HPK and HPK-tainer systems utilize the same pass-through scheduler and custom Virtual Kubelet implementation to bridge the control plane with the compute nodes, ensuring the fundamental mechanics of task orchestration are preserved. However, the Virtual Kubelet now submits jobs directly to the bubble via bash instead of submitting them to Slurm.

Deployment is tightly integrated with Slurm through a batch script that identifies the allocated nodes, selects a controller node, launches bubbles using srun, and configures the temporary overlay network for the duration of the job. From the HPC platform perspective, deployment only requires Apptainer (run with –fakeroot and –network none), slirp4netns, and support for user namespaces. In contrast with previous designs, the HPC host only sees one active job (the entire cluster), while all internal K8S jobs run completely invisibly to the host system administrators.

4.3.1 HPK-2.0 Evaluation

We validate HPK-tainer in a proof-of-concept environment by deploying K3s in the controller bubble, using a modified HPK Virtual Kubelet to manage workload containers, and using Flannel as the Container Network Interface (CNI). The evaluation deployed an Apache Dask application consisting of one scheduler and distributed worker containers, which communicated through the HPK-tainer overlay network while executing a distributed matrix-multiplication workload.

The workload generated random matrices of 4,000×4,000 elements and partitioned them into blocks of 500×500 elements for distributed processing. Across the evaluated runs, the cluster completed the computation in an average of 34.42 seconds. Mean CPU utilization remained low and stable, averaging 3.4%, while peak CPU utilization reached 68.6%, depending on the run.

These results demonstrate that HPK-2.0 can support distributed, peer-to-peer, cloud-native workloads within a batch-scheduled HPC environment. In particular, the successful execution of the Dask scheduler and workers confirms that workload containers deployed across the allocated environment can establish the network communication required for coordinated, multi-stage processing.

5. Multisite and Data-Aware Workflow Execution

This section presents the multisite architecture developed for data-aware federated workflow execution. The architecture addresses the dual challenges of enabling seamless execution across heterogeneous Kubernetes sites and implementing an intelligent scheduling policy to reduce the movement of Copernicus original products. To achieve this, the section covers the following architectural pillars:

- Federated Control Plane: We establish a Kubernetes federation using Karmada to manage multiple member clusters as a single entity. Importantly, since Argo Workflows does not natively support multi-cluster dispatch through Karmada, we developed the necessary bridge logic to allow the workflow engine to orchestrate pods across the entire federation.

- Artifact-Native Pilots: Beyond the infrastructure layer, we refactored the pilot workflows to utilize an artifact-native pattern. By moving away from site-local shared filesystems and treating all inputs and outputs as artifacts, we ensure we can move workflows between sites.

- Proof-of-Concept Site Jumps: We demonstrate the "site jump" capability for specialized hardware. In our proof-of-concept, we validate that the scheduler can successfully route GPU-heavy sub-graphs to a remote cluster when the local site lacks necessary accelerators, using a shared MinIO instance to maintain the artifact plane.

- Data-Aware Scheduling with Rucio: Finally, we describe the metadata publication process to the Rucio catalogue. This enables our scheduler to identify the location of durable Copernicus assets across the federation, allowing it to perform data-aware scheduling that prioritizes data locality while making informed, cost-based exceptions for performance-critical site jumps.

5.1 Multisite Workflow Orchestration

Running workflows across a single Kubernetes cluster becomes limiting as datasets become larger and applications become more computationally expensive. Furthermore, different sites usually differ in ownership, hardware, and governance. A naive response to this constraint would be to push the burden onto the people writing workflows, expecting data scientists to manually decide which site to run each step on, manage data placement between facilities, and reason about compliance and data residency rules. This approach fails in practice because it forces domain experts to take on operational responsibilities far outside their expertise. Additionally, it introduces risks of human error, inefficient placement, and violations of data residency requirements that no individual user should be expected to track manually across every job they submit. There is also a structural problem beneath the operational one: distributed facilities are rarely under a single owner's control, and connecting sites that sit under different governance structures means respecting each site's autonomy while still making the resources behave as one coherent system rather than a set of isolated islands. A multicluster architecture is what resolves both problems at once, since it removes placement, resource management, and data engineering decisions from the user entirely, letting workflows span whichever combination of clusters, domains, and even non-Kubernetes HPC environments a computation actually needs, without the user ever having to specify where a given step should run.

Karmada is the component that makes this abstraction possible at the Kubernetes level. It is an open-source multi-cluster orchestration project that gives operators a single, Kubernetes-native control plane for deploying and managing workloads across many clusters, whether those clusters live on-premises, in the cloud, or in a hybrid mix of both. Rather than requiring a user to pick a target cluster explicitly, Karmada applies policy-driven scheduling, so that where a workload lands is determined automatically by rules covering resource availability, hardware requirements, and cluster health, while the underlying

architecture of a master control plane and per-cluster agents keeps every member cluster registered and monitored centrally. This is what allows the platform to treat clusters in different locations, including ones exposing scarce hardware such as GPUs, as a single elastic pool of compute rather than separate silos a user has to reason about individually.

In DaFab, we extend the multisite concept by one more layer towards the end-user, by integrating Argo workflow engine on top of the container orchestration substrate. Combining Karmada with Argo Workflows carries this same principle down to the level of individual workflow steps: Argo continues to define and track the lifecycle of a workflow exactly as it would in a single cluster. Simultaneously, Karmada possesses the technical capability to intercept individual step pods and distribute them across member clusters based on resource or hardware availability. The default execution policy for DaFab prioritizes scheduling the complete per-product workflow at a single site. This intended placement strategy ensures that data-intensive intermediates remain local, using cross-cluster pod placement only as a deliberate exception when specific hardware requirements cannot be met locally.

5.1.1 Multisite Kubernetes with Karmada

Karmada [13] is an open-source, CNCF incubating control plane that overlays a set of independent Kubernetes clusters and exposes them as one logical environment. At its front door sits karmada-apiserver, essentially a repackaged kube-apiserver, paired with an aggregated extension that offers the Cluster API through karmada-aggregated-apiserver and registers Karmada specific API groups such as policy.karmada.io. This duo keeps the surface identical to vanilla Kubernetes while letting the project add multicluster primitives without forking client tools. Scheduling decisions are taken by karmada-scheduler, a standalone process that watches for new objects, evaluates placement rules, and writes the resulting ResourceBinding objects back to the API. Continuous reconciliation is split between the kube-controller-manager, which runs stock controllers against the control plane host cluster, and the karmada-controller-manager, whose multicluster controllers turn each binding into per cluster Work manifests.

Cluster registration can be achieved with two methods. The first is push mode, where the Karmada plane sends workloads to members directly: the operator supplies the member cluster's kubeconfig, Karmada stores the credentials in the generated Cluster object, and karmada-controller-manager pushes Work directly to the member's kube-apiserver. The second is pull mode, where members poll for workloads from the Karmada plane instead: the operator creates a short lived bootstrap token, then deploys a lightweight karmada-agent inside the member cluster, and this agent watches karmada-apiserver, fetches Work, and applies it locally. In both cases a Cluster custom resource is created in Karmada's etcd. Karmada's etcd and karmada-webhooks are components similar to the Kubernetes native ones, but they exist exclusively for Karmada specific objects.

Once the fleet is assembled, multicluster delivery is a two step exercise. As shown in Figure 5.1, the author writes an ordinary Kubernetes object, such as a Deployment, and alongside it a PropagationPolicy or its cluster scope variant, which expresses placement intent covering eligible clusters, selectors, taints and tolerations, replica splitting, geo spread, and similar

concerns. Optional fine tuning goes into an OverridePolicy, which tells Karmada how to patch the workload per target cluster, such as changing an environment variable only in EU regions or shrinking CPU requests on edge nodes. When these resources are applied to the Karmada API, the scheduler resolves the policy set, produces a ResourceBinding that enumerates target clusters and replica counts, and the propagation controllers materialize the workload inside each member cluster while continuously reconciling drift. Karmada's policy based approach lets users manage clusters spread across regions as if they were a single system. They declare where and how each workload should run, and Karmada then places replicas predictably, maintains high availability, and handles cross-region failover automatically, with no custom scripts required.

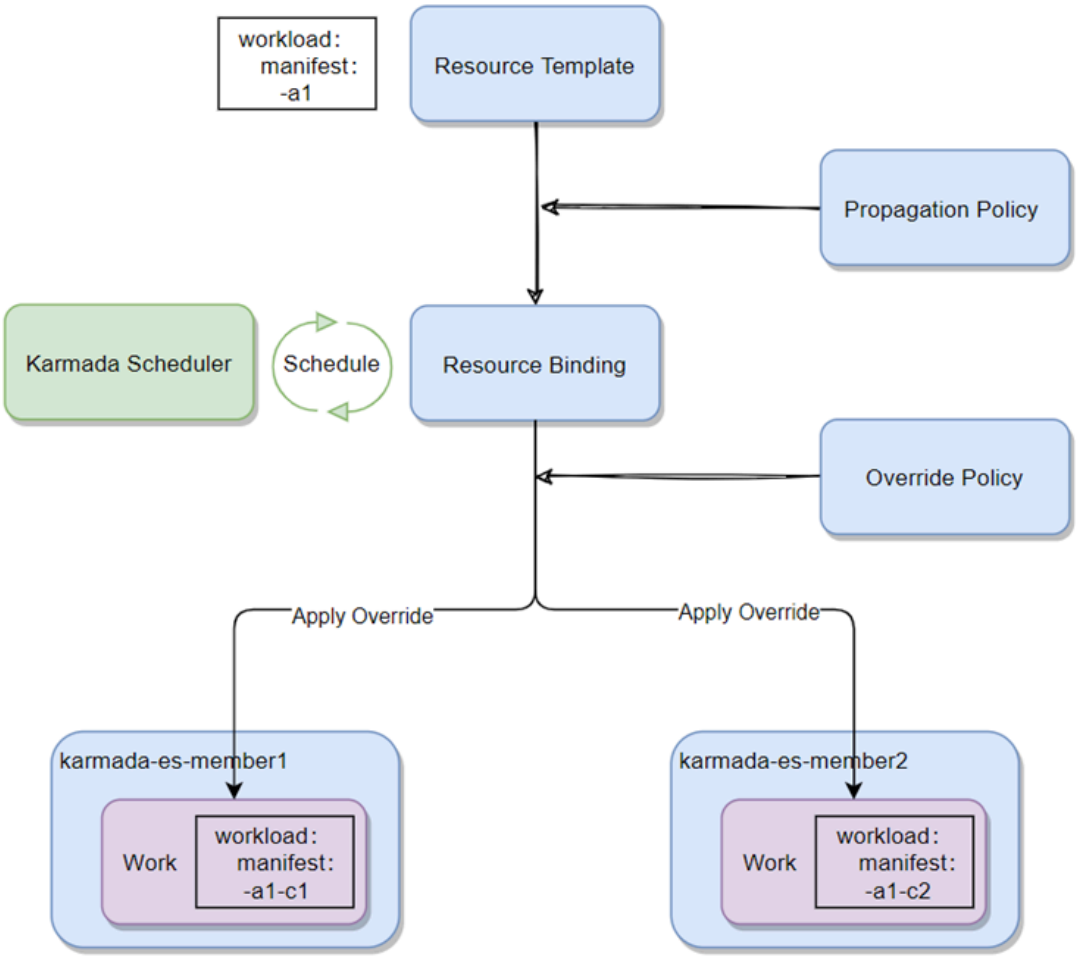


Figure 5.1: Karmada Policies guide the selection of member clusters for workload execution.
Image source: https://karmada.io/docs/key-features/features

The DaFab multicluster proof-of-concept deployment consists of two Kubernetes clusters, Gcore and Forth, registered as host-cluster and remote-cluster within a single Karmada

federation. This setup specifically serves to validate the "site jump" capability: Gcore provides the primary CPU and memory resources for data-heavy stages, while Forth contributes a dedicated GPU node for specialized AI acceleration. By federating these complementary compute profiles, the proof-of-concept demonstrates how the system can automatically transition a workflow subgraph from one site to another when local hardware is insufficient. Forth joins the federation using Karmada's pull-based registration mechanism—a natural choice for navigating stricter network boundaries, as it enables the remote site to participate in the federation without requiring inbound connectivity from the central control plane.

5.1.2 Implementation of Multisite Workflow Orchestration

Argo [14] is an open-source, Kubernetes-native workflow engine that orchestrates jobs. It represents each job as directed acyclic graph where each node represents a discrete unit of work. Kubernetes represents a workflow as a custom resource named *Workflow*. The Argo Workflows controller running in the cluster watches for these objects and drives them through to completion, translating each step in the graph into a pod that runs the corresponding container image and command. Steps may depend on one another explicitly, so that a step only starts once its declared dependencies have succeeded, and data can be passed between steps through parameters or artifacts, so that outputs produced by one step are available as inputs to the next regardless of which node it eventually runs on. Templates form the reusable building blocks of a workflow: a Workflow specification is composed of one or more templates, each describing either a container to run, a script to execute, or a composition of other templates, which lets complex pipelines be assembled from smaller, testable pieces rather than a single monolithic definition. Once submitted, the controller continuously reconciles the actual state of the pods it has created against the desired state described in the Workflow object. It retries failed steps according to configured policies, enforces timeouts, and marks the overall workflow as succeeded or failed once every step has been resolved.

The main idea behind combining Argo Workflows and Karmada is to separate workflow definition from workflow placement, letting each system do the part it is actually designed for. Argo Workflows defines a pipeline as a graph of steps and manage each step's lifecycle, retries, and data dependencies. However, it has no native concept of a federation of clusters and assumes every pod it creates lands on the single cluster it is pointed at. Karmada, conversely, is built entirely around deciding where a workload should run across a fleet of clusters, but it has no notion of a multi-step pipeline or the dependencies between steps; it only understands discrete Kubernetes objects to be scheduled. To bridge this gap we enhance Argo controller to use the Karmada API instead of a plain Kubernetes API: Argo keeps authoring and sequencing the workflow exactly as it would in a single cluster, while every pod it generates is transparently intercepted by Karmada and placed on whichever member cluster satisfies the policy in effect, whether that is the cluster with the most free CPU and memory or the one exposing a GPU. This enhanement of Argo is essential because splitting a workflow across clusters manually, by hardcoding which step runs where, would eliminate the flexibility of dynamic scheduling and force every workflow author to

reason about cluster capacity and hardware placement themselves rather than letting the platform absorb that responsibility.

Making this bridge work in practice is difficult because Karmada's control plane is not designed to run third-party controllers itself. As a result, Argo Workflows components must be hosted on an ordinary cluster and pointed at the Karmada API rather than a local Kubernetes API. This immediately introduces a credentials and CRD synchronization problem. The Argo controller needs an admin-level kubeconfig for the Karmada API to create and monitor Workflow objects and their generated pods across the federation. That kubeconfig has to be delivered to the controller as a mounted secret with the controller's deployment explicitly reconfigured to use it instead of its in-cluster identity. A second, more subtle issue is that Argo's CRDs and RBAC rules only make a member cluster capable of running workflow pods; they do not make it capable of reporting on them. Specifically, the workflowtaskresults CRD, which Argo's executors use to report per-step status, has to be present and correctly permissioned on every member cluster expected to run steps, and any cluster missing it will run the pod but fail to report progress back to the controller, leaving the workflow appearing stuck even though the underlying container executed correctly. Because this requirement applies per member cluster, it becomes a recurring setup cost every time a new cluster joins the federation rather than a one-time configuration step.

In the current DaFab proof-of-concept, we address cross-cluster orchestration by hosting the Karmada-facing Argo Workflows controller directly on Gcore alongside the Karmada control plane. This instance is isolated in a separate namespace from the Knot-facing Argo instance to avoid interference. To simplify management, Karmada’s resource propagation mechanism automatically pushes the required Argo CRDs, ServiceAccounts, and RBAC bindings to the Forth cluster upon registration. This transforms what would otherwise be a manual, error-prone setup into a declarative, automated process. Forth joins the federation via pull-based registration, initiating contact with the Karmada control plane using a bootstrap token, which simplifies credential management and avoids the need for inbound connectivity from Gcore.

Regarding data access, the current deployment utilizes a shared central MinIO S3 instance hosted on Gcore. All workflow pods, regardless of whether they are scheduled on Gcore or Forth, read and write artifacts to this central store. This design choice was made for the proof-of-concept to decouple data availability from compute placement, thereby demonstrating the platform's ability to schedule steps purely based on hardware requirements (e.g., GPU availability) without being blocked by data locality constraints.

However, this centralized model serves as a functional bridge to the final data-aware architecture. The system first queries Rucio to determine location of the Copernicus products. If the local site (where the data resides) lacks the necessary hardware for a specific step, the scheduler evaluates the GPU site jump: it will only trigger a remote execution at the remote site if the AI acceleration benefit is projected to exceed the data transfer and packaging overhead. By establishing this artifact-native pattern now, we ensure the workflow engine is already architected to handle data as explicit, addressable objects, enabling the cost-aware scheduling of Task 4.2 to choose between pinning the workflow for locality or jumping for performance.

The combination of these three elements defines the architecture: a Karmada control plane on Gcore federating two member clusters with different compute profiles, a dedicated Karmada-facing Argo Workflows instance separated from the Knot-facing one, and a single centrally reachable MinIO instance that removes data locality as a constraint on where any given step can run. This architecture (Figure 5.2) integrates Argo Workflows for lifecycle management with Karmada for cross-cluster resource scheduling. The execution sequence begins when a user externally submits a workflow manifest via the Argo Workflows UI, which forwards it to the Argo Workflows Controller. As the workflow progresses, the controller continuously interacts with the Karmada API Server, allowing the Karmada Scheduler to dynamically bind workload pods to their designated target clusters sequentially. Workloads are distributed across cluster boundaries based on hardware requirements, meaning that Workflow CPU Step 1 and Workflow CPU Step 3 execute locally within the Member host-cluster, while Workflow GPU Step 2 is offloaded to the Member remote-cluster for hardware acceleration. Finally, a MinIO S3 storage instance deployed within the host-cluster provides a shared data layer, allowing both Step 2 and Step 3 to read and write intermediate artifacts seamlessly across these distinct cluster environments. How the Karmada scheduler actually decides where each step is placed, and the challenges that placement logic introduces, is covered in the next section.

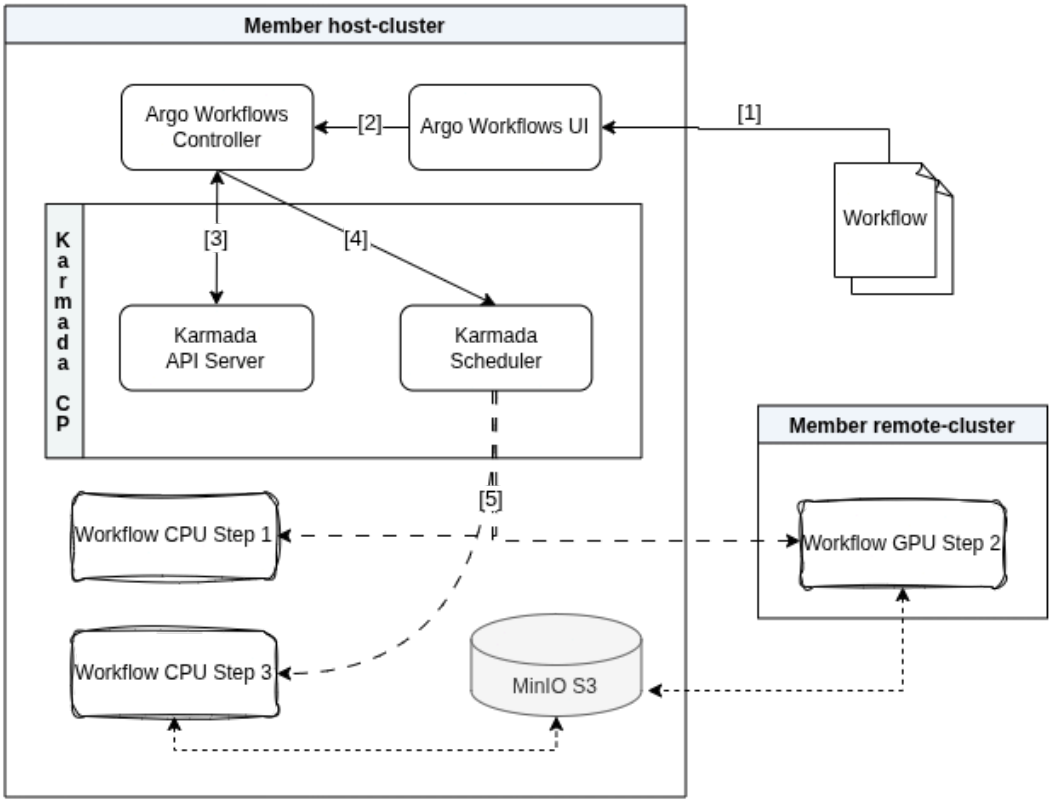


Figure 5.2: Karmada Multicluster Architecture

Consider a simple three-step workflow: a preprocessing step, a training step that needs GPU acceleration, and a postprocessing step that consumes the training output. Under a default

resource-balanced policy, all three steps would be treated identically and placed purely according to which member cluster reports the most free CPU and memory at submission time, which is adequate for the preprocessing and postprocessing steps but wrong for the training step if Forth is the only cluster exposing a GPU. To route the training step correctly, its pod template carries a label such as accelerator: gpu, and a second PropagationPolicy targets pods bearing that label specifically, using a cluster affinity constrained to remote-cluster and a higher priority value than the default policy so it is evaluated first. Any pod without the label falls through to the resource-balanced policy exactly as before, while any pod carrying the label is pinned to the GPU-bearing cluster regardless of that cluster's current CPU and memory headroom.

```yaml
apiVersion: policy.karmada.io/v1alpha1
kind: PropagationPolicy
metadata:
  name: gpu-steps
spec:
  resourceSelectors:
  - apiVersion: v1
    kind: Pod
    labelSelector:
      matchLabels:
        accelerator: gpu
  placement:
    clusterAffinity:
      clusterNames:
      - remote-cluster
```

This works cleanly as long as the label is known at workflow authoring time, but it breaks down the moment the decision of whether a step needs a GPU depends on something only knowable at runtime, such as an input parameter, a value computed by a previous step, or the size of a dataset resolved from an upstream artifact. Static labels written into the workflow spec cannot react to that kind of information, since the spec is fixed before the workflow starts executing. This is where a mutating admission webhook extends the same mechanism into runtime scheduling: Argo's step pods pass through the webhook as they are created, and the webhook inspects fields available at that point, for example an annotation set by a prior step or a parameter substituted into the pod spec, and injects the accelerator: gpu label onto the pod object itself before it is persisted. Because Karmada's PropagationPolicy selection happens after admission, the same gpu-steps policy picks up the newly injected label without any change to the policy itself, meaning the placement logic stays entirely declarative and the only new component introduced is a small webhook that decides, at the moment each pod is actually created, whether that particular instance of the step qualifies for GPU placement.

5.1.3 Breaking Data SILOs of Multisite Workflows via Artifacts

Designing workflows for a federated environment requires a fundamental shift from local-state assumptions to explicit, artifact-based data management. A workflow designed for a single cluster typically assumes that intermediate files written by one step are immediately available to the next via shared local storage (e.g., NFS or local PVs). However, in a Karmada federation, consecutive steps of the same Directed Acyclic Graph (DAG) may be scheduled on entirely different clusters. Relying on local storage in this context leads to data invisibility and workflow failures, as a step placed on a remote cluster cannot access the local volume of its predecessor.

To resolve this, workflows in DaFab adopt an artifact only approach. In this approach, all intermediate data is treated as explicit artifacts stored in an object storage layer (S3/MinIO) or filesystem that is globally reachable across the federation. This approach decouples data availability from compute placement, ensuring that any workflow step can retrieve its inputs and store its outputs regardless of the physical cluster where it executes. This transition from implicit filesystem handoffs to explicit artifact declarations is the cornerstone of DaFab's multisite strategy, enabling true workload portability.

The migration of an existing Earth Observation pipeline (a 16-step DAG) to this architecture highlighted the necessity of this shift. The original pipeline relied on implicit NFS paths, which are incompatible with cross-cluster scheduling. By migrating to Argo-native artifacts, each step now declares its inputs and outputs with complete S3 connection details. This ensures that workflows are self-contained and portable, capable of executing on any member cluster without requiring pre-configured shared storage paths.

During this migration, we address specific technical challenges in cross-template artifact resolution. For instance, ensuring reliable extraction of directory artifacts across cluster boundaries required standardized packaging (e.g., .tgz formats) to preserve metadata integrity. These refinements ensure that the workflow engine can robustly handle data handoffs in a heterogeneous environment, removing the fragility associated with local state dependencies.

The adoption of an artifact-native model unlocks two distinct levels of scheduling optimization:

1. Hardware-aware Placement (Current proof-of-concept): with data location no longer a hard constraint, the scheduler is free to place steps based purely on compute requirements. The current DaFab PropagationPolicies leverage this freedom to optimize for hardware fit. Steps are dynamically routed to clusters with the appropriate resources—for example, CPU-intensive processing to high-core-count nodes, and GPU-intensive inference to accelerator-equipped nodes. This ensures that specialized hardware is utilized efficiently across the federation, regardless of where the data originated.

2. Data-Aware Placement (Target Architecture): the explicit tracking of data as artifacts also lays the foundation for locality-aware scheduling. Because every Copenicus origina product artifact is registered in the Rucio catalogue, the system has the

visibility needed to determine where data resides. This allows the system to pull compute to data, reducing data transfers. The current artifact-native design ensures that when this locality-aware policy is activated, the workflow engine will already be prepared to handle data as explicit, locatable entities rather than implicit local files.

In summary, the current implementation demonstrates the portability and hardware-optimization capabilities of the DaFab architecture. By establishing this artifact-based data layer, we ensure that the execution engine is decoupled from storage-specific paths. However, workload portability is only the first half of the challenge; the second half is providing the scheduler with the intelligence to know where these data entities actually reside. To move from run anywhere to run where the data is, the federation needs a global view of data distribution. This visibility is provided by the integration of DASI and Rucio, which transform the isolated site-local caches into a unified, cooperative caching registry.

5.2 Data Aware Scheduling with DASI and Rucio

While the artifact-native design described in the previous section makes workflows portable, the coordination between DASI (Data Access and Storage Interface) and Rucio is what makes the scheduler data-aware. This layer acts as the federation's data intelligence system, mapping logical Copernicus products to physical cluster locations.

DaFab Earth Observation workflows access Copernicus products through the Data Access and Storage Interface (DASI) by ECMWF. In the current production path, the orchestration layer supplies DASI with a specific Sentinel product identifier and the requested asset keys. DASI checks its local archive and if the assets are missing, automatically acquires the necessary metadata and assets from the Copernicus Data Space Ecosystem (CDSE). Once retrieved, DASI archives the data, flushes the store to ensure persistence, and validates the assets through a read-back process.

Crucially, once these assets are successfully cached and validated at the site, the system informs Rucio of their new location. This registration ensures that the federation-wide catalogue is updated in real-time, allowing the scheduler to recognize the site as a valid data-resident location for future workflow placements. This identifier-based approach ensures that the workflow executes against precise, granular assets while maintaining the global visibility required for data-aware orchestration.

This separation is important because DASI is responsible for accessing and populating local storage, whereas Rucio acts as the authoritative discovery and replica catalogue. The workflow infrastructure does not need to embed storage paths or site-specific access assumptions in workflow definitions. Instead, it can resolve a logical dataset identifier through Rucio and obtain the locations at which the required data is already available.

The same principle applies to the management of intermediate workflow artifacts. For each per-product workflow, the scheduler identifies a primary eligible site where the original Copernicus data resides and pins the execution there to ensure that intermediates remain local. However, when a "site jump" is required—such as routing a GPU-heavy sub-graph to a remote cluster—the system specifically configures the intermediate artifacts of that sub-graph to be accessible from the remote site.

This catalogue-driven approach supplies the data-location information required by data-aware scheduling. For a workflow requiring multiple input products, the scheduler can determine the fraction of the requested dataset already present at each candidate site and use this information together with resource availability, accelerator requirements, transfer cost, and data-residency constraints to select an execution location. This directly supports the DaFab objective of scheduling a workflow at the Copernicus site holding the largest fraction of its requested dataset.

5.2.1 Transfer Cost Modeling

This section outlines the elements for a transfer cost model policy intended to guide data-aware scheduling decisions in DaFab. The model selects the best data source for each processing task by evaluating the following parameters that reduce the estimated end-to-end time-to-result.

A given Copernicus product may reside in one or more locations: in one or more remote DASI caches across the federation's sites and at the central Copernicus datahub. For each task, there are two execution strategies:

1. CPU-based execution: If the task requires only CPU resources (preprocessing or postprocessing), the policy selects the site that hosts a copy of the input data and offers the best estimated completion time. This estimate includes:
    a. Transfer time from the data source to the site. It is zero if already local, otherwise a function of time to first byte (TTFB), bandwidth, and dataset size.
    b. Estimated CPU processing time (P_cpu, provided by WP2)
2. GPU-based execution: If the task benefits from GPU acceleration (e.g., AI inference step), the policy evaluates two options:
    a. Local GPU execution: Run the task at a site that both hosts a copy of the data and has available GPU resources.
    b. Remote GPU execution with staging: Stage the data to a GPU-enabled site if no local GPU site holds a copy. The total time estimate includes:
        i. Transfer time from the nearest data source (cached or origin) to the GPU site.
        ii. Estimated GPU processing time (P_gpu, provided by WP2)

In both cases, Rucio contains the information to discover all known locations of the required dataset and retrieves the latest TTFB and bandwidth estimates for each source to target path. Site jumps are only triggered when: a GPU-accelerated task cannot run locally and staging to a remote GPU site yields a lower time-to-result than running CPU-only locally, or explicit priority or deadline constraints favor remote execution.

5.2.2 Scheduling of Workflows

Argo Workflows natively executes every step of a workflow as a pod inside a single Kubernetes cluster, which becomes a constraint once a platform grows beyond one cluster's capacity or needs to reach hardware, such as GPUs, that is only available in specific clusters. To address this, Argo Workflows was combined with Karmada, a multi-cluster orchestration layer built on the native Kubernetes API, so that individual workflow steps

rather than the whole workflow could be distributed across clusters. Karmada treats each connected member cluster similarly to how a single Kubernetes scheduler treats nodes, evaluating placement decisions centrally and then dispatching the resulting work objects to the chosen cluster.

The step-level split across clusters highlighted the limitations of local persistent storage (such as NFS), which is tied to a single cluster's nodes. Since workflow steps could now land on different physical clusters, local storage was no longer sufficient for passing intermediate data. We address this challenge by adopting an artifact-native approach, where intermediate data is stored in an object storage layer (S3/MinIO) reachable from all member clusters. This design decouples data access from specific cluster infrastructure, ensuring that any step can retrieve its inputs regardless of where it is scheduled.

With data accessibility ensured, the focus shifts to intelligent placement. Karmada manages this through PropagationPolicy objects, which determine the target cluster for each workflow step. In the current proof-of-concept, two layered policies are implemented:

1. Resource-Aware Placement (Default): For the majority of steps, Karmada's 'Schedule based on Cluster Resource Modeling' is used. This feature models each cluster's available CPU and memory using a customizable resource grid. Steps are propagated to the cluster with the most available headroom, ensuring load balancing across the federation.
2. Hardware-Aware Placement: For steps requiring specific hardware (e.g., GPUs), a second policy takes precedence, pinning these workloads to clusters that expose the necessary resources (such as the FORTH GPU node).

While the current implementation prioritizes resource efficiency and hardware matching, the underlying artifact-native architecture lays the groundwork for data-aware scheduling. By treating data as explicit artifacts, the system is prepared to integrate locality metrics (via Rucio) in the next phase, allowing the scheduler to prefer sites where the required data already resides, thus minimizing transfers.

The default path applies to the majority of steps and relies on Karmada's Schedule based on Cluster Resource Modeling capability. This feature lets Karmada model each member cluster's available CPU and memory according to a customizable resource grid rather than relying on generic node capacity assumptions, and it is what drives Karmada's divided scheduling behavior when spreading replicas across clusters with uneven headroom. In this setup, steps without special hardware requirements are propagated to whichever cluster currently reports the most free CPU and memory according to that model, keeping the load balanced across the federation as steps are continuously created and completed.

The resource model itself is defined in grades, each one specifying a CPU and memory range that a cluster's remaining capacity must fall into to be classified at that grade. Four grades were configured, from grade 0 covering clusters with less than 1 CPU core and 2Gi of memory free, up through grade 3 covering anything above 4 CPU cores and 8Gi of memory free, with the upper bound left effectively unbounded to absorb clusters with large amounts of spare capacity. This grading gives Karmada a discrete, comparable measure of headroom across clusters rather than forcing it to reason about raw, continuously varying

resource numbers, which matters once the federation grows beyond two clusters and headroom differences become less obvious to reason about manually.

```
resourceModels:
  - grade: 0
    ranges:
    - max: "1"
      min: "0"
      name: cpu
    - max: 2Gi
      min: "0"
      name: memory
  - grade: 1
    ranges:
    - max: "2"
      min: "1"
      name: cpu
    - max: 4Gi
      min: 2Gi
      name: memory
  - grade: 2
    ranges:
    - max: "4"
      min: "2"
      name: cpu
    - max: 8Gi
      min: 4Gi
      name: memory
  - grade: 3
    ranges:
    - max: "9223372036854775807"
      min: "4"
      name: cpu
    - max: "9223372036854775807"
      min: 8Gi
      name: memory
```

The propagation policy that consumes this model, named resources-balanced-pods, selects all Pod resources, which in this context are the pods generated per workflow step by Argo, and considers both host-cluster and remote-cluster as eligible targets through its cluster affinity. Its replica scheduling is set to Divided with a replicaDivisionPreference of Aggregated, which instructs Karmada to send each pod to a single cluster rather than splitting replicas across multiple clusters, and to prefer the cluster with the most aggregated free capacity according to the grades above. A priority of 1 places this policy as the baseline behavior for any step pod that is not otherwise claimed by a higher priority policy such as the GPU based one.

```
apiVersion: policy.karmada.io/v1alpha1
kind: PropagationPolicy
metadata:
```

```
  name: resources-balanced-pods
spec:
  resourceSelectors:
  - apiVersion: v1
    kind: Pod
  priority: 1
  placement:
    clusterAffinity:
      clusterNames:
      - host-cluster
      - remote-cluster
    replicaScheduling:
      replicaSchedulingType: Divided
      replicaDivisionPreference: Aggregated
```

The second path handles steps that require GPU acceleration, mainly AI inference or training steps embedded in the workflow. These steps are labeled at the workflow specification level, and a separate propagation policy uses that label to target only the clusters that expose GPU nodes, bypassing the general resource-based placement entirely. Because the GPU policy is selected through the label selector, it takes precedence for any step carrying the GPU label, while every unlabeled step falls back to the default CPU and memory based policy. This label-driven precedence keeps the two policies from conflicting, since a given step is only ever eligible for one of them at a time.

The net effect is a two-tier scheduling model layered on top of Karmada: hardware-aware placement for GPU-bound steps, and resource-aware load balancing for everything else, both operating transparently underneath the workflow definition so that Argo Workflows itself remains unaware that its steps are being redirected across a federation of clusters rather than run on a single one.

To provide a complete overview of the workflow scheduling and system architecture, the following two diagrams illustrate the structural differences between the single-cluster NFS setup and the multi-cluster solution using Argo Artifacts.

Diagram 1 (Figure 5.3) traces how a workflow step submitted through Knot acquires its storage before any container starts. The user's request enters Argo through Knot's backend, which attaches the appropriate NFS-backed volume definitions to the step before the workflow controller ever creates a Pod. Once the controller hands that Pod spec to the Kubernetes API and a node is chosen, the NFS CSI driver takes over, binding the PVC to a PV and mounting the underlying NFS share onto the node before the kubelet pulls any container images. Only after that storage is in place does the kubelet begin pulling images for the init and main containers and starting the step. The diagram deliberately excludes artifact handling so the storage path stands on its own: submission, scheduling, volume binding, mount, then image pull and execution, with status flowing back up to the controller once the Pod completes.

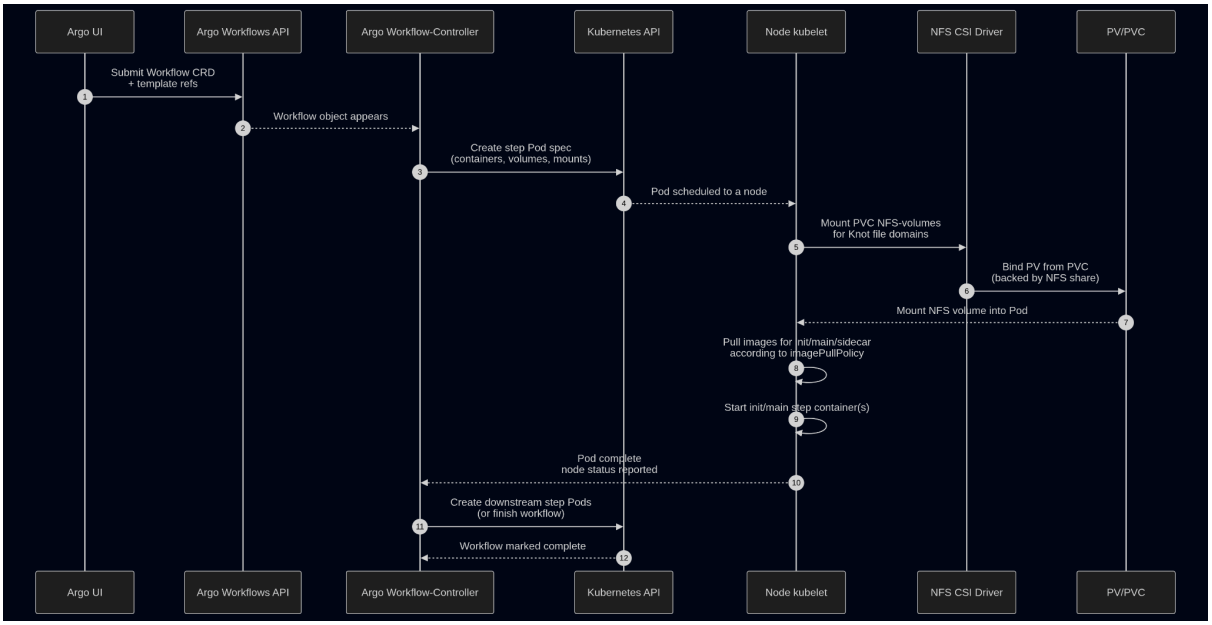


Figure 5.3: Single Kubernetes cluster with NFS storage

Diagram 2 (Figure 5.4) with Karmada propagation extends that same submission path but changes who owns scheduling once Argo hands off the Pod. Rather than the workflow controller talking directly to a single cluster's API, it submits the step's Pod spec to the Karmada API server, which evaluates the applicable PropagationPolicy and lets the Karmada scheduler pick a member cluster for that specific step. Karmada's controller manager then creates the Pod in the chosen cluster, at which point the flow rejoins the original pattern locally within that member cluster, image pulls, init executor downloading input artifacts, main container execution, and the sidecar executor uploading output artifacts to the repository. What changes is the return path: status does not go straight back to Argo but is reported up through the member cluster to Karmada's controller manager and aggregated at the Karmada API before the workflow controller sees it, and this same negotiation repeats independently for each subsequent step, so different steps of one workflow can end up running in different clusters.

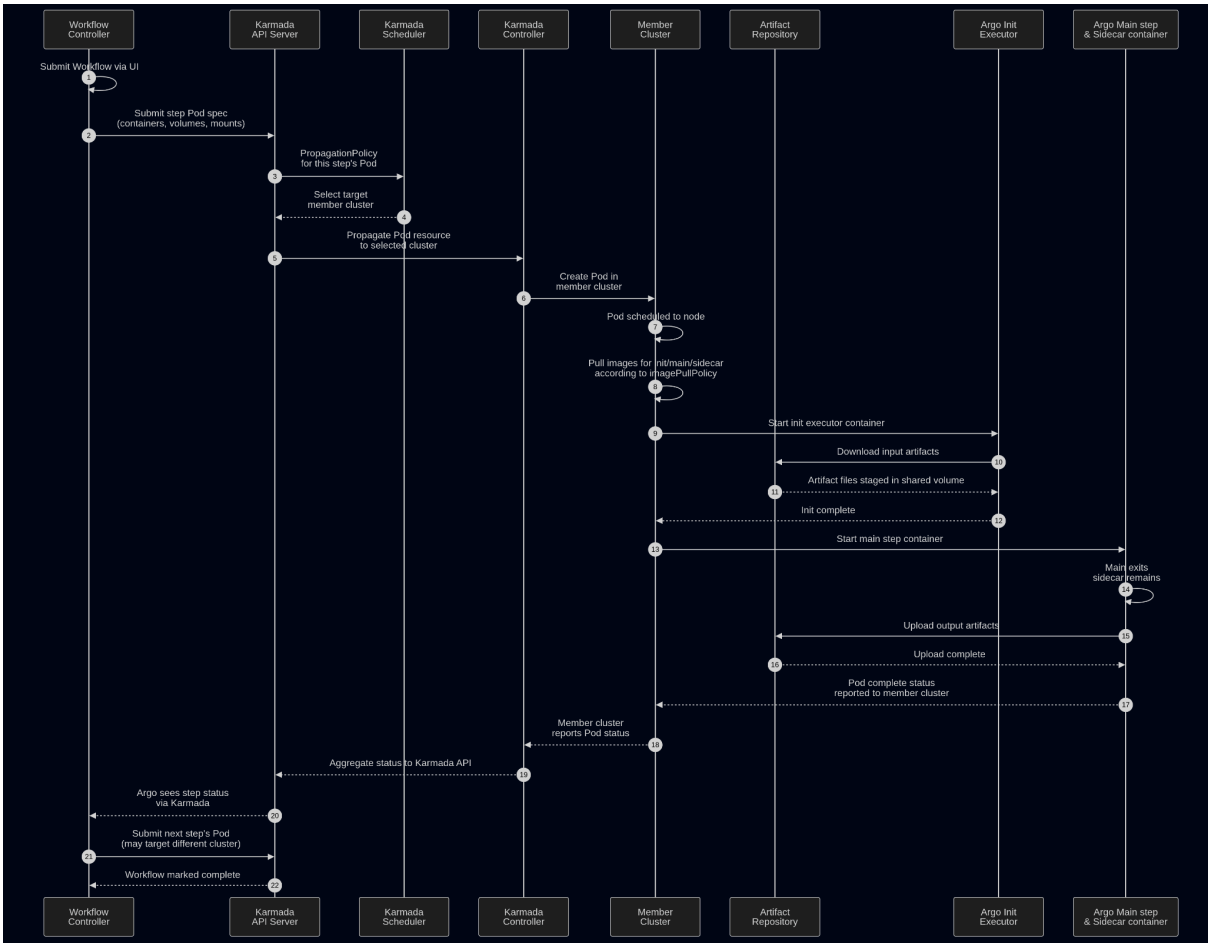


Figure 5.4: Multiple Kubernetes clusters with Argo Artifacts

6. Conclusions

Deliverable D4.2 defines a reference architecture for federated, data-aware Earth Observation workflows in the DaFab project. By combining HPK-2.0, Karmada-based multi-cluster orchestration, and artifact-native workflows, the proposed approach bridges Cloud and HPC environments while reducing dependence on site-specific storage systems. The validation activities demonstrate that workloads can be executed flexibly across distributed European infrastructures, including specialised computing resources. This provides a strong basis for addressing data-gravity challenges in Copernicus processing. Future work will build on this architecture by introducing locality-aware scheduling policies that align computing tasks with data availability, improving efficiency, scalability, and cost-effectiveness across the federation.

Bibliography

[1] A. Tzenetopoulos et al. (Ed.). (2022). EVOLVE: Towards Converging Big-Data, High-Performance and Cloud-Computing Worlds. *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, *2022*, 975-980. 10.23919/DATE54114.2022.9774698

[2] Chazapis, A., Maliaroudakis, E., Nikolaidis, F., Marazakis, M., & Bilas, A. (2024). Running Cloud-native Workloads on HPC with High-Performance Kubernetes. *arxiv*. https://arxiv.org/abs/2409.16919

[3] https://datashim.io/. (2025). *Datashim*.

[4] McCrory, D. (2010). Data gravity: In the clouds. Thépaut, J.-N., Dee, D., Engelen, R., & Pinty, B. (2018). The Copernicus Programme and its Climate Change Service. *IEEE International Geoscience and Remote Sensing Symposium*. 10.1109/IGARSS.2018.8518067

[5] Petsis G., Chazapis A., Marazakis M., Bilas A. (2026). Running Containers in HPC Inside Private, User-Level Network Overlays. ICS Workshops '26. *dl.acm.* https://dl.acm.org/doi/10.1145/3774895.3813789

[6] Chazapis A., Vassilakis L., Petsis G., Marazakis M., Bilas A. (2025). Evaluating HPK for Running Cloud-Native Workloads on Slurm Clusters. SC Workshops '25. *dl.acm.* https://dl.acm.org/doi/10.1145/3731599.3767352

[7] Flannel. (2026). Flannel: A network fabric for containers. *GitHub.* https://github.com/flannel-io/flannel

[8] Calico. (2026). Calico: Open-source networking and network security for containers, virtual machines, and bare metal hosts. *Github.* https://github.com/projectcalico/calico

[9] Apptainer. (2026). Apptainer. https://apptainer.org/

[10] etcd. (2026). etcd: A distributed, reliable key-value store. https://etcd.io/

[11] k3s. (2026). K3s: Lightweight Kubernetes. https://k3s.io/

[12] slirp4netns. (2026). slirp4netns: User-mode networking for unprivileged network namespaces. GitHub. https://github.com/rootless-containers/slirp4netns

[13] Karmada (2026). Karmada: Multi-Cluster Kubernetes Orchestration https://karmada.io/

[14] Argo Workflows (2026). Argo Workflows: https://argoproj.github.io/workflows/

[15] European Space Agency. (2026). Copernicus Sentinel-2 [level/product name] [Data set]. Retrieved from https://dataspace.copernicus.eu/

DaFab Consortium